An Empirical Study of Experience Replay for Control in Continuous State Domains

by

Xin Li

A thesis submitted in partial fulfillment of the requirements for the degree of

Master of Science

Department of Computing Science

University of Alberta

© Xin Li, 2022

Abstract

In this thesis, we investigate the empirical performance of several experience replay techniques. Efficient experience replay plays an important role in model-free reinforcement learning by improving sample efficiency through reusing past experience. However, replay-based methods were largely forgotten before being repopularized by the Deep Q-Learning Network (DQN) architecture and have since become a standard component in offline training. In this work, we revisit understudied classic replay strategies, backward replay and on-policy replay, a heuristic for replaying trajectories in the temporally backward order following on-policy action sequences, proposed in the original experience replay literature. We re-evaluate them in several classic reinforcement learning control problems under linear and nonlinear function approximations. We observe that (1) on-policy replay outperforms off-policy replay in two continuous state 2D maze domains under an exploratory policy, and (2) contrary to the previous claim in the original replay literature, replay settings exist where on-policy replay underperforms off-policy replay. We hypothesize that the practical benefit of on-policy replay is problem dependent and sensitive to the state distribution of the replay buffer. In addition, we propose a simple time-stepping replay strategy called "jumpy replay" that takes advantage of state generalization to speed up value propagation, which presents a comparable or better performance with a vanilla backward replay baseline across 5 replay settings.

To my parents.

The world of reality has its limits; the world of imagination is boundless.

– Jean-Jacques Rousseau.

Acknowledgements

This thesis would not be possible without the many brilliant people I have been privileged to know and learn from. I am incredibly fortunate to work under the guidance of Professor Martha White and Adam White. They have been a constant source of inspiration and kindness, and I will be forever grateful. I sincerely acknowledge and appreciate their contribution in proposing the ideas and helping me develop the experiments involved in this project. Their approach to research and life has immeasurably influenced me and helped me grow as an RL researcher. I would like to thank my incredible teammate and collaborator, Han Wang, for always being supportive and helpful when working together through a difficult time. Finally, I would like to acknowledge and thank my amazing friends and mentors, Raksha Kumaraswamy and Andrew Jacobsen, for generously guiding me with their expert advice and helping me proofread, edit and polish this report.

Contents

1 Introduction **1**
1.1 Objective 4
1.2 Contributions 4

2 Background Material **6**
2.1 MDP 6
2.2 Value Function 7
2.2.1 Q-learning 7
2.2.2 Linear Function Approximation with Tile coding . . . 8
2.2.3 Nonlinear Function Approximation with Artificial Neural Network 10
2.3 Experience Replay 12

3 An Overview of Experience Replay **13**
3.1 Backward Replay 16
3.2 Backward Replay with n-step Targets 17
3.3 Biased Backward Replay 17
3.4 Unbiased Backward Replay 18

4 Experiment Design **21**
4.1 Environments 21
4.2 Experimental Setup 25
4.3 Function Approximation 26
4.4 Evaluation Metric 27

5 Evaluation of unbiased replay **29**
5.1 Linear Function Approximation 31
5.2 Nonlinear Function Approximation 34
5.3 Analysis of Agent Performance and Replay Step Budget . . . 43
5.4 Conclusions 44

6 Jumpy Replay **45**
6.1 Empirical Results 47
6.1.1 Linear Function Approximation 48
6.1.2 Nonlinear Function Approximation 49

7 Conclusion and Future Work **51**

References **54**

List of Figures

2.1 Illustration of how tile coding representations are generated . . 8
2.2 Illustration of neural network function approximation 10

3.1 Illustration of classic experience replay with online data collection. To the left is an agent under a behaviour policy π_k interacting with the world. To the right are the experience replay buffer and the replay process consisting of an online and an offline update procedure. 15
3.2 Policies found by q-learning agents without replay, with random replay, and backward replay halfway through the second episode. The blue arrows indicate the greedy actions in each state. If an arrow is not shown in a state, then its action values are equal. 16

4.1 Continuous Gridworld 21
4.2 Puddle World 23
4.3 Lunar Lander 24

5.1 Learning curves of agents with linear function approximation in ContGW. 31
5.2 Replayed States of 50-backward-step agents with linear function approximation in ContGW (averaged per thousand timesteps). 32
5.3 Learning curves of agents with linear function approximation in PuddleWorld. The unbiased replay agents converge to the cumulative reward of about -45. 33
5.4 Replayed States of 50-backward-step agents with linear function approximation in PuddleWorld (averaged per thousand timesteps). 33
5.5 Learning curves of agents with nonlinear function approximation in ContGW. 35
5.6 Replayed States of 50-backward-step agents with nonlinear function approximation in ContGW (averaged per thousand timesteps). 35
5.7 Learning curves of agents with nonlinear function approximation and an exploratory policy in PuddleWorld. 36
5.8 Replayed states of 50-backward-step agents with nonlinear function approximation and an exploratory policy in PuddleWorld (averaged per thousand timesteps) 37
5.9 Learned values of 50-backward-step agents with nonlinear function approximation and an exploratory policy in PuddleWorld 38
5.10 Learning curves of agents with nonlinear function approximation and a greedy policy in PuddleWorld. 38
5.11 Learned values of 50-backward-step agents with nonlinear function approximation and a greedy policy in PuddleWorld. . . . 39

5.12 Replayed states of 50-backward-step agents with nonlinear function approximation and a greedy policy in PuddleWorld 40
5.13 Learning curves of agents with nonlinear function approximation in LunarLander. 42
5.14 Average return under a different number of replayed trajectories per timestep. 43
5.15 Average return under different number of backward steps. . . 44

6.1 An example of time-stepping mechanic in jumpy replay. The solid line depicts a suboptimal trajectory with detours. The dashed line shows the concept of jumpy replay where a predefined number of k timesteps are skipped between each replay step. 47
6.2 Learning curves of 10-backward-step jumpy replay agents with linear function approximation. 48
6.3 Learning curves of 10-backward-step jumpy replay agents with nonlinear function approximation. 49

Chapter 1

Introduction

Reinforcement learning (RL) is a problem formulation for experience-driven autonomous learning, where an intelligent agent interacts with an environment and learns to act. Instead of being told which actions to take, it requires only a scalar reinforcement signal as performance feedback from the environment. It consists of a diverse set of subproblems and a class of classic solution methods such as Temporal Difference (TD) learning [32]. More recently, the advent of Deep Learning (DL) has accelerated progress in the study of many problems in machine learning and drastically improved state-of-the-art vision, and language tasks such as object classification, speech recognition, and language generation [9]. Compared to the previous generation of Artificial Intelligence (AI), the immense success of DL can be largely credited to its ability to automate representation learning and is often able to discover compact low-dimensional features from high-dimensional and often unstructured data (e.g., audio, text, and images). Similarly, much of the success of Deep RL has been built upon scaling classic RL techniques to problems that require learning from multimodal high-dimensional data. As the most notable example, Mnih et al. [24] first demonstrated that a single learning algorithm can be applied to play a diverse set of Atari 2600 video games, directly learn from raw image pixels, and achieve a superhuman level performance. It was soon followed by many other successes, such as AlphaGo, a Deep RL agent that defeated a human world champion in Go for the first time in history [29].

Two important subproblems of RL include temporal credit assignment, and

generalization [34]. The latter is also known as the structural credit assignment problem [20]. In recent years, much progress has been made in solving the generalization problem epitomized by a better understanding of representation learning and the development of deep neural network architectures. In comparison, temporal credit assignment studies the problem of understanding the relationship between actions and rewards, which could sometimes occur after an arbitrarily long period of delay [34]. For example, given a sequence of states and actions an agent experienced in an environment, an agent needs to have the ability to correctly assign credit or blame to some state-action pairs to learn an optimal behaviour through interactions. One of the most popular solution methods to the problem is the temporal difference (TD) learning [31], which is closely connected to dynamic programming [2].

Popular planning techniques used in dynamic programming (DP) where a generative model of the Markov Decision Process (MDP) is accessible include value iteration and policy iteration [3]. Typical DP other model-based planning assume a model of the environment. Alternatively, in the model-free setting, experience replay methods are used to perform asynchronous backups. Experience replay is a technique where an agent's online experience is stored as state transition tuples in memory, to be sampled later and used to update the value function in a separate asynchronous planning process. Better understanding of replay, where real experience is used to improve an agent's policy, could likely help develop insights that can be applied in model-based planning, where generated experience is used for policy improvement.

Sample efficiency is a key performance metric that measures the number of environment interactions an agent needs to obtain a good policy. Designing a sample-efficient agent remains a major challenge in RL. Online Q-learning algorithms are often inefficient in that experiences obtained by trial-and-error are utilized to adjust the value function only once and then thrown away. This can be expensive and wasteful since some experiences may be rare or costly to obtain [20]. Experience replay was then developed to improve sample efficiency by repeatedly performing TD-style updates using trajectories sampled from a buffer of recent agent-environment interactions. In addition, experience replay

can be used to help break down correlations in updates between temporally sequenced transitions an online agent would experience [23]. Furthermore, with function approximation in consideration, imperfect models are susceptible to compounding error; real experience is typically more informative than generated ones [33].

Replay-based methods have been largely forgotten before being repopularized again by Mnih et al. [23]. A variant of the experience replay mechanism was introduced to ease the training of a Deep Q-Learning Network (DQN) agent by addressing the problem of correlated data and smoothing the behaviour distribution for mini-batch sampling. As a comparison, the classic replay mechanism in Lin [19] iterates through transitions on a sampled trajectory in the temporally backward order and performs correlated updates, while DQN-style replay samples mini-batches of state transition tuples usually in an independent and identically distributed (i.i.d.) fashion for the value function update. Lately, there have been many extensions in the use of experience replay, and most have been built upon the DQN-style replay. Schaul et al. [28] investigated how prioritizing some transitions over others in a replay buffer could make replay more efficient by reusing experiences with high expected learning progress. Zhang and Sutton [43] proposed a new DQN-style replay method that remedied the negative influence of a large buffer by appending an online experience to mini-batches. Jafferjee et al. [13] suggested that updating values of real states towards values of unreachable states results in misleading state-action values that adversely affect the agent's policy.

Lately, there have been more efforts to understand the replay mechanism. For example, Van Hasselt, Hessel, and Aslanides [36] showed that under some conditions, a model-free agent with DQN replay is competitive with a model-based agent variant given less experience and computation in Atari 2600 video games. Fedus et al. [8] studied how DQN-style replay affects the performance of deep reinforcement learning agents and found that n-step returns are crucial for taking advantage of larger buffer sizes. However, the community has spent many efforts studying DQN-style replay while the classic experience replay was largely overlooked.

Our study aims to explore this understudied topic and gain insights into classic experience replay through empirical study. Our motivation is threefold. First, as the original proposal of reusing past experience, classic experience replay is unstudied relative to DQN-style replay. Its empirical performance using function approximation remains to be thoroughly examined. Second, insights into classic experience replay might help inform future research related to planning, a process that replays simulated experience generated by a model. Third, there are signs that classic experience replay could be biologically motivated as evidence shows that a similar replay mechanism in the reverse temporal sequence has been observed in the brain [6, 1]. A better understanding of classic replay could help inspire further development in related domains in neuroscience research.

1.1 Objective

In this thesis, we revisit some old ideas for experience replay. In particular, we perform an empirical study of understudied replay strategies, backward replay and unbiased replay, which were first proposed in Lin [19]. We conduct experiments under a modern training regime similar to DQN to compare the sample efficiency of select replay methods and provide empirical insights on choosing backward replay algorithms that are more effective at performing temporal credit assignment when function approximation is used.

1.2 Contributions

This thesis summarizes an empirical study comparing the sample efficiency property of replay techniques under standard RL training regimes, more specifically,

1. we perform an empirical investigation of Lin's classic experience replay design, backward replay, and unbiased replay using modern function approximation and optimizers. We conduct a comprehensive evaluation over select design dimensions in the backward replay, the number of

timesteps to skip between updates and whether to perform trajectory backups in an unbiased v.s. biased fashion, using three classic continuous state domains, and present results extending beyond previous observations in Lin [19]. Note we explore these design dimensions under the context of backward trajectory replay, similar to Lin's original proposal.

2. Based on experiment results, we have found that unbiased replay outperforms off-policy backward replay in two continuous state 2D maze domains under an exploratory policy. Furthermore, we are able to identify two exceptions where unbiased replay underperforms and provide some analysis. We also hypothesize that the potential benefit from using unbiased replay is problem dependent and sensitive to the choice of behaviour policy.

3. We propose a replay strategy called jumpy replay that takes advantage of state generalization to speed up value propagation, which demonstrates a comparable or better performance against a backward replay baseline in 5 replay settings.

Chapter 2

Background Material

This chapter provides some background knowledge necessary to understand the experiments in later chapters. We begin with introducing the problem setting and its formulation, followed by reviewing the concept of value functions, different types of function approximations, and experience replay.

2.1 MDP

The environments in RL are typically framed as Markov Decision Process (MDP), a framework modelling a discrete-time process. The MDP is a quadruple $\mathcal{M} := (\mathcal{S}, \mathcal{A}, \mathcal{P}, \mathcal{R})$, where $\mathcal{S}$ and $\mathcal{A}$ are the state and action space. $p : \mathcal{S} \times \mathcal{R} \times \mathcal{S} \times \mathcal{A} \to [0, 1]$ is the transition probability, which encodes the dynamics of the MDP, $P(s', r|s, a)$. It represents the conditional distribution of transitioning to state s' and emitting a reward of r, from state s, upon taking action a. At each timestep $t = 1, 2, ...$, the environment is in a state $S_t \in \mathcal{S}$, an agent observes it and takes an action $A_t \sim \pi(\cdot|S_t)$, where $\pi : \mathcal{S} \to \mathcal{A}$ is the policy learned by the agent. The environment transitions the agent to a next state $S_{t+1} \sim \mathcal{P}(\cdot|S_t, A_t)$ and emits a reward $R_{t+1} \in \mathbb{R}$ determined by the reward function $r : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \to \mathbb{R}$. The agent aims to maximize its expected return, which is the expected total reward obtained in the long term, defined by $G_t \coloneqq R_{t+1} + \cdots + R_T$ where T refers to the final timestep. Agents in real-world problems often face the challenge of the curse of dimensionality and require function approximation as a result. In this thesis, we will focus on the MDP setting with continuous state space and finite action space.

2.2 Value Function

Almost all RL methods involve estimating value functions, which are functions of states or state-action pairs that estimate how good a state s or an action a is in order to maximize an agent's total rewards in its lifetime. The state value function, denoted by $v_\pi(s)$, under policy π, estimates the expected total return given a state $s \in S$. It is defined as

$$v_\pi(s) \coloneqq \mathbb{E}_\pi[G_t|S_t = s]. \tag{2.1}$$

The optimal policy is defined over all states as

$$v_*(s) \coloneqq \max_\pi v_\pi(s). \tag{2.2}$$

In addition, an action-value function $q(s, a)$ is used to estimate the expected return an agent could expect upon taking an action a in state s. In a similar fashion, the policy-dependent action value function and the optimal action value function are defined respectively as,

$$q_\pi(s, a) \coloneqq \mathbb{E}_\pi[G_t|S_t = s, A_t = a] \tag{2.3}$$

and

$$q_*(s) \coloneqq \max_\pi q_\pi(s, a), \tag{2.4}$$

where the expectation is taken with respect to a policy π, under the transition dynamics provided by the MDP. The policy from the optimal value function π_* is an optimal policy.

2.2.1 Q-learning

Q-learning [39] is an algorithm that iteratively updates learned action value function of state-action pairs $Q(s, a) : s \times a \to \mathbb{R}$ towards the optimal action values $q^*(s, a)$, where q^* is defined as the action value under an optimal policy. The action value estimates are updated iteratively following an update rule,

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha[R_t + \gamma \max_a Q(S_{t+1}, a) - Q(S_t, A_t)], \tag{2.5}$$

where $R_t + \gamma \max_a Q(S_{t+1}, a) - Q(S_t, A_t)$ is called the temporal difference error (TD-error) and α is the step size of an update that affects the learning speed of the agent. The resulting fixed point $Q(S_t, A_t)$ satisfies the following Bellman optimality equation,

$$Q^*(s, a) = \mathbb{E}[R_{t+1} + \gamma_{t+1} \max_{a' \in \mathcal{A}} Q^*(S_{t+1}, a') | S_t = s, A_t = a]. \tag{2.6}$$

2.2.2 Linear Function Approximation with Tile coding

When the state and action spaces are small enough, a value function can sometimes be simply represented in a table; however, in many real-world problems, the state space is often large or continuous. Therefore, it would be useful to use a function approximator to approximate the state value function $q^{\pi}_{\mathbf{w}}(s, a)$, where $\mathbf{w}$ represents the parameters of the value function. In addition, limited computational resources is another consideration when many RL practitioners choose to apply function approximation in value space.

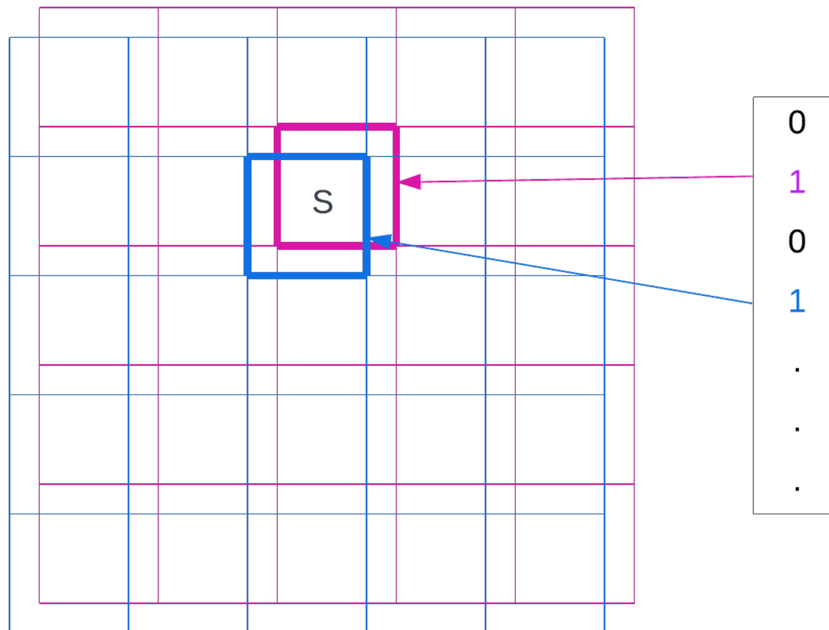

Figure 2.1: Illustration of how tile coding representations are generated

Linear function approximation is the simplest form of function approxima-

tion method, where value function is parameterized by a linear combination of the state features $\phi(s) \in R^d$, and we are interested in learning $\mathbf{w} \in R^d$, where d represents the dimension of the feature space. The value functions parameterized by $\mathbf{w}$ are given by

$$\hat{v}(s, \mathbf{w}) = \mathbf{w}^T \phi(s) \tag{2.7}$$

and

$$\hat{q}(s, a, \mathbf{w}) = \mathbf{w}^T \phi(s, a). \tag{2.8}$$

Q-learning with linear function approximation follows the semi-gradient update rule,

$$\mathbf{w}_{t+1} = \mathbf{w}_t + \alpha(R_{t+1} + \gamma \max_a \mathbf{w}_t^T \phi(s_{t+1}, a) - \mathbf{w}_t^T \phi(S_t, A_t))\phi(S_t, A_t) \tag{2.9}$$

There're many different approaches to constructing a feature vector for a state-action pair, such as Fourier basis, Krylov basis, radial basis functions, and coarse coding [15, 27, 21]. While the linear approximation scheme may seem naive for approximating a complicated value function, its capacity could be enhanced by using feature construction techniques to craft non-linear state action feature encoding. Tile coding, a special case of coarse coding, is one powerful way to encode state features in multi-dimensional continuous state space. In tile coding, receptive fields of the features are partitioned into different groups. Each group is referred to as a tile, and each partition is called a tiling. Since each partition divides the state space into non-overlapping tiles, an input feature would be assigned an active tile for every partition applied as part of the predetermined configuration. The resulting state representation is an n-hot binary encoding of the raw observation, where n is the number of tilings. Figure 2.1 shows a simple example to demonstrate tile coding's feature construction. In addition, we could also choose to tile code each dimension of the raw input features independently or together based on the nature of an environment. Tile coding has been shown to successfully solve challenging RL tasks [30].

2.2.3 Nonlinear Function Approximation with Artificial Neural Network

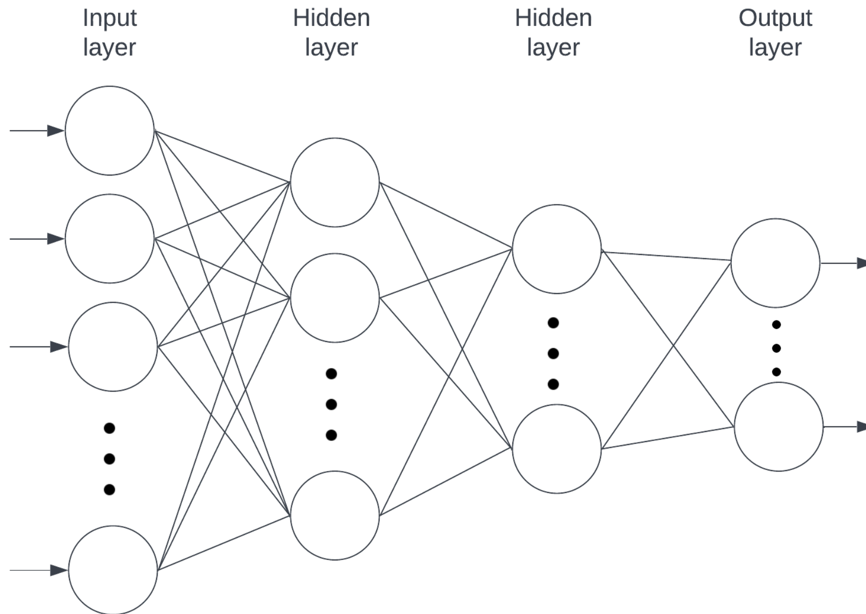

Figure 2.2: Illustration of neural network function approximation

Linear function approximation has led to many successes in studying classic RL problems in the past. However, custom feature engineering has largely remained a part of the training process that requires non-negligible human efforts, especially for multimodal or raw pixel inputs [17]. In an effort to promote end-to-end training following the success of deep learning and expand the application of existing RL techniques to modern environments, Mnih et al. [24] proposed the Deep Q-Network (DQN) architecture, which is a type of non-linear function approximation that combines the learning of features and the value function simultaneously. DQN uses a deep neural network consisting of multiple convolutional and perceptron layers to learn a representation from raw pixel inputs and approximate the value function. It was the first to achieve human-level performance in the Atari 2600 simulator environment with raw inputs only. The update for a DQN network is as follows,

$$\mathbf{w}_{t+1} = \mathbf{w}_t + \alpha(R_{t+1} + \gamma \max_a \tilde{q}(s_{t+1}, a, \mathbf{w}_t) - \hat{q}(S_t, A_t, \mathbf{w}_t))\nabla_w \hat{q}(S_t, A_t, \mathbf{w}_t), \tag{2.10}$$

where $\tilde{q}$ is the value function parameterized by the target network, which will be explained in a later section.

Online v.s. i.i.d. Samples

One of the challenges for training deep RL agents with mini-batch data is the inherent temporally correlated nature of agent experience, but often, the convergence of deep learning algorithms relies on the assumption of i.i.d. samples. An online agent using function approximation would tend to overfit to more recent experiences that likely only cover a small part of the whole state space and impairs control performance of the task. To mitigate the impact of correlation in mini-batch training, DQN reintroduced the experience replay mechanism and used the uniform sampling technique to construct a mini-batch for training deep neural nets. Mnih et al. [22] also suggested a distributed data collection method that utilizes multiple agents to interact with multiple environments simultaneously to acquire temporally uncorrelated data. This thesis will mainly focus on exploring the use of sampled partial trajectories when training a value network, which often contains temporally correlated state transitions.

Target Network

Target network is another new technique introduced in DQN. Training deep RL agents suffers from instability due to the deadly triad, which occurs when bootstrapping, function approximation, and off-policy learning are being used simultaneously [32]. The concept of a target network, denoted by $\tilde{q}$, was proposed to reduce the chance of divergence. It is a copy of the network in training $\hat{q}$ used in computing the bootstrapping target, which periodically updates its parameter values in sync with the network in training. Even with a target network, stability remains an issue as soft divergence has been observed in DQN agent training [35].

2.3 Experience Replay

The central idea of experience replay is to reuse past experiences to improve the sample efficiency of a learning algorithm. It was first proposed by Lin [19] and more recently adapted by Mnih et al. [23]. Lin [19] introduced the classic style experience replay mechanism as part of an offline training procedure.

The classic experience replay takes the form of a memory that stores episodes of agent experience. After each episode, a number of episodic trajectories would be sampled from memory. For each sampled episode, state transitions are visited in the temporal backward direction, and the value function of the agent is updated using the Q-learning update rule. In comparison, Mnih et al. [23] described an online variant of experience replay. After each timestep, the last encountered transition tuple (s, a, r, s) is stored into a replay buffer following the first-in-first-out fashion. A mini-batch of state transitions is sampled uniformly from the memory to update the value function in an i.i.d. fashion. Most of the recent works in deep RL followed a similar design of experience replay [28, 38, 18]. In this thesis, we will refer to this style of experience replay as the DQN-style replay.

In contrast to the DQN-style replay, we will focus on an online variant of the classic experience replay setup in this study, which performs correlated updates instead of using i.i.d. samples. We maintain a running buffer of state transition tuples given a certain capacity. The oldest entry will be removed if the memory is full. At each timestep, we begin with training the agent using the online transition. It is followed by a replay procedure, where a number of partial trajectories are sampled in the outer loop and each trajectory is replayed in the backward order. Finally, Q-learning updates to the value function are performed with respect to each state transition tuple in the inner loop. We will further discuss this variant of replay in the next section.

Chapter 3

An Overview of Experience Replay

Classic experience replay was developed as a way to speed up credit propagation and shorten an agent's trial-and-error process [20]. It was a simple mechanic that trains an agent in the background with quadruples of state transitions $(s, a, r, s^{'})$. In the original design, the state transitions are stored in the form of episodic experience in a sliding window buffer in a first-in-first-out fashion. The agent is trained offline every so often after an episode ends when several recent episodes are sampled from the buffer for replay. An episode consisting of a sequence of experience tuples is replayed in temporally backward order. We refer to this method as backward replay in this chapter. On top of that, Lin advocated that only state transitions that had taken on-policy actions based on current value estimates should be replayed during backward replay. We will refer to this selective replay approach as on-policy replay in the rest of the thesis. We will elaborate on these two replay strategies in the following sections.

There are several benefits to using experience replay when training an RL agent. First, some online experiences can be rare to encounter during exploration due to the MDP or invokes a large penalty for doing so. Saving these rare experiences for replay later could be useful. As intuition behind information theory suggests, rare events are generally more surprising or uncertain. Learning from an unlikely event is usually more informative than learning that a likely event has occurred [12]. Second, the replay mechanic plays to

the advantage of stochastic optimization and mini-batch training. This can be very useful, especially when training a deep Q-network. In deep learning, multiple-pass over training data with stochastic gradient descent (SGD) has generally shown faster convergence than a single-pass for training deep neural networks both empirically and theoretically [42]. Similarly, learning a good value function using stochastic optimization methods likely benefits from visiting an experience multiple times since SGD typically requires a small step size for an update. Experience replay naturally facilitates the need for experience reuse and stochastic sampling. Last, trajectory replay is a simple and direct approach to solving the temporal credit assignment problem, especially when some credit or blame must be propagated through a long sequence of actions for an agent to learn a good policy. However, the use of experience replay comes with a cost simply from the extra memory needed to store experiences. Another limitation is that past experiences may become irrelevant or even harmful, especially when an environment is highly non-stationary; therefore, recent experiences tend to be more useful for replay than experiences from distant past [16, 20]. For example, in Lin's original design, the buffer only stores the last 100 episodes of agent experience [19].

We would like to explore the impact of design choices introduced in the original replay under a modern RL context. More specifically, we adapted the classic experience replay to a modern RL setting that differs from Lin's original design in several aspects. First, our study focuses on measuring an online agent's sample efficiency. In this setting, whether an online experience is included in the value function update could impact the agent performance and potentially become a confounding factor in our study, as previously shown in Zhang and Sutton [43]. As a solution, our agent performs a value function update using the online experience at every timestep, in addition to the replay loop. Second, we chose to use a large state transition buffer instead of a smaller episodic memory of recent agent experience as in Lin's design, since a state transition buffer is more commonly used in deep RL today. Empirical insights using a large state transition buffer could be more relatable to real-world applications, where transition buffer is a popular choice and memory is

relatively cheap. As a result of using a state transition buffer, we modified the replay loop to sample a starting state transition and replay a fixed number of steps specified by a hyperparameter, where Lin's original replay samples an entire episode for replay each time. Also, note the original replay is a pure offline process where the replay loops happen after a new episode terminates, while we take an online approach to replay where a number of replay loops start after the online update at each timestep. Last, the original replay design utilized a recency-biased sampling strategy where more recent episodes are exponentially more likely to be chosen [20]. Even though we have studied the effect of recency bias in the past [16], we stay with uniform random sampling in this study to reduce confounding factors in our empirical results. The complete steps of our adapted version of classic experience replay are shown in Figure 3.1.

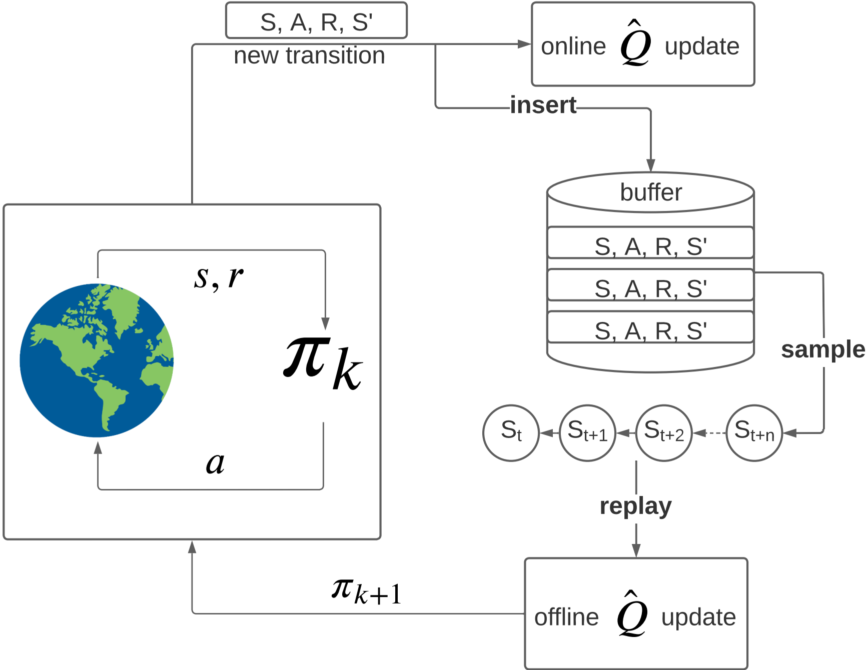

Figure 3.1: Illustration of classic experience replay with online data collection. To the left is an agent under a behaviour policy π_k interacting with the world. To the right are the experience replay buffer and the replay process consisting of an online and an offline update procedure.

3.1 Backward Replay

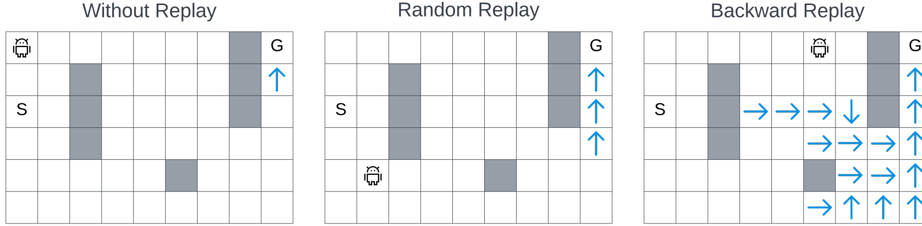

Figure 3.2: Policies found by q-learning agents without replay, with random replay, and backward replay halfway through the second episode. The blue arrows indicate the greedy actions in each state. If an arrow is not shown in a state, then its action values are equal.

TD learning is a slow process for temporal credit assignment, especially when reward signals have to be propagated through a long action sequence [20]. In physiology, replay mechanism in temporal sequences has been observed in the brain. Studies have linked it to experience replay in model-free reinforcement learning and shown that it could be used to improve learning compared to no replay [6]. Furthermore, correlations between backward replay and changes in reward have been discovered in animal experiments more recently [1]. Replay following the reverse temporal order has also been proposed under reinforcement learning and dynamic programming frameworks several times [19, 7]. The central idea is that experience replay can be more effective in propagating credit if a sequence of experiences is replayed in temporally backward order. This general idea is termed *backward focusing* of planning computations [32].

Algorithm 1 Backward Replay

1: Loop repeat n times:
2: Sample a trajectory : $S_0, A_0, R_1, \ldots, S_T, A_T$
3: **for** $i = T-1, T-2, \ldots, 0$ **do**
4: $\quad \delta_i = R_{i+1} + \gamma max_a \tilde{q}(S_{i+1}, \cdot, \mathbf{w}) - \hat{q}(S_i, A_i, \mathbf{w})$
5: $\quad$ Update $\mathbf{w}$ with $\nabla(\delta_i^2)$ for a step size η
6: **end for**

3.2 Backward Replay with n-step Targets

Backward replay naturally enables the use of n-step return targets similar to a Monte-Carlo method, which is often used to speed up temporal credit assignment despite its higher variance compared to a TD(0) target. On top of it, an agent can choose to perform value backup along an on-policy or an off-policy trajectory in the planning loop. In this section, we give a more detailed review of this choice, analyze its pros and cons, then consider a trade-off between the unbiased (on-policy) and biased (off-policy) backup methods.

Intuitively, n-step returns represent the "credit" to assign, and how a replay method selects partial trajectories for backing up valuable information helps decide which states are being assigned said "credit" or "blame". One-step return (λ-return with a λ of 0) were used in Lin's earlier work [19, 20]. In our experiments, we introduce a modern addition of growing n-step return along a trajectory to obtain $G_{t:t+n}$ in each update rather than lambda returns. This allows us to easily compute the n-step return in its recursive form and effectively perform credit assignments over different timescales as we replay backward without explicitly choosing the time interval over which bootstrapping is done,

$$G_{t:t+n} = r_{t+1} + \gamma r_{t+2} + ... + \gamma^n \hat{v}(s_{t+n}, \mathbf{w}_{t+n-1}) \tag{3.1}$$

3.3 Biased Backward Replay

The use of experience replay in Q-learning with a neural network used for value function approximation was first studied in [19, 20]. It proposed the idea of sample reuse as some experiences may be rare and sometimes costly to obtain. The original work used the recursive form of the lambda return while replaying backward from the end of an episode with Monte-Carlo TD updates [8]. This type of multi-step backward replay facilitates a longer planning horizon to be considered. In our study, we would like to incorporate a modern addition, which is n-step return. To do so, we will consider two replay vari-

ants, an uncorrected and biased replay method and one that replays on-policy trajectories and updates using unbiased n-step returns.

Unlike the unbiased backup, being more intuitive in Dyna and other model-based settings [32, 26], as it simply involves generating trajectories for backups with the help of a model using the same policy as behaviour policy, performing biased backups is more straightforward compared to unbiased backup when an experience replay is used. Biased backup propagates value information along directly sampled trajectories in a buffer generated by recent policies and uses the updated action values to guide an agent's online policy, which resembles policy iteration. More specifically, on every timestep after an agent encounters a real experience and performs an online TD update, the planning process loops through several sampled trajectories and performs TD updates along these trajectories in the reverse temporal order using growing n-step returns.

Algorithm 2 Biased Multi-step Replay

1: Loop repeat n times:
2: Sample a trajectory : $S_0, A_0, R_1, \ldots, S_T, A_T$
3: Initialize bootstrap target: $U \leftarrow max_a \tilde{q}(S_T, \cdot, \mathbf{w})$
4: **for** $i = T-1, T-2, \ldots, 0$ **do**
5: $\quad U \leftarrow R_{i+1} + \gamma U$
6: $\quad \delta_i \leftarrow U - \hat{q}(S_i, A_i, \mathbf{w})$
7: $\quad$ Update $\mathbf{w}$ with $\nabla(\delta_i^2)$ for a step size η
8: **end for**

3.4 Unbiased Backward Replay

One notable choice in multi-step replay is when to stop and restart backups. A practical replay strategy would be to sample a starting state, replay a moderate number of steps backward following a trajectory, then resample a new starting state and repeat the process. Conceptually, each value propagation step in Q-learning could be broken down into two steps following the bellman backup [37],

$$q_\pi(s,a) \leftarrow r(s) + \gamma \sum \pi(a|s) \sum P(s'|s,a) v_\pi(s') \tag{3.2}$$

$$v_\pi(s) \leftarrow max\ q_\pi(s,a) \tag{3.3}$$

Lin [19] proposed that we could stop replay early on off-policy actions to prevent wasteful updates. The intuition here is that an update to an off-policy action value of a state s does not affect its maximum action value. The state value estimate $\hat{v}(s, \mathbf{w})$ stays the same, and so does the new behaviour policy $\pi_b(a|s)$. As a result, replaying off-policy actions will not be as helpful when our goal in control tasks is to improve the behaviour policy with fewer updates. Here we argue that whether we replay backward only following on-policy actions makes a difference in the types of updates we end up performing. Note in control, the target policy is a greedy policy with respect to the optimal value function $q*$, and an on-policy action is defined as $a_{on} \leftarrow argmax\ q_\pi(s,a)$. If the sampled partial trajectory follows the target policy, it is a partial optimal trajectory. Performing backups following a partial optimal trajectory are equivalent to a small backup [37] of value iteration in dynamic programming since a greedy action is taken with respect to $\hat{q}*$. In comparison, biased backups introduced earlier propagate value information over transitions regardless of its action being on-policy, which is more comparable to sample backups under policy iteration in effect. Unbiased backward replay attempts to actively focus value propagation along partial trajectories in high probability regions under the on-policy distribution in the hope of speeding up temporal credit assignment.

Intuitively, a greedy action has a larger action value than other actions of a state by definition; thus, an update of a greedy action value is more likely to induce a larger update. In turn, such an update is more likely to improve the policy quickly. Additionally, if the current state action is suboptimal, then the implied state value $v_\pi(s)$ would only be affected through generalization after an update with respect to a suboptimal action value $q_\pi(s, a_{\text{off}})$. If we continue to replay backward after such an update, it is likely of little help. The original ER paper studied this problem and proposed two on-policy replay methods, AHCON-R and QCON-R [20], that are pure Monte-Carlo methods. Here we

use a modernized version of the approach with the addition of n-step returns.

Algorithm 3 Unbiased Multi-step Replay

1: Loop repeat n times:
2: Sample a trajectory : $S_0, A_0, R_1, \ldots, S_T, A_T$
3: Initialize bootstrap target: $U \leftarrow max_a\tilde{q}(S_T, \cdot, \mathbf{w})$
4: **for** $i = T-1, T-2, \ldots, 0$ **do**
5: $\quad U \leftarrow R_{i+1} + \gamma U$
6: $\quad \delta_i \leftarrow U - \hat{q}(S_i, A_i, \mathbf{w})$
7: $\quad$ Update $\mathbf{w}$ with $\nabla(\delta_i^2)$ for a step size η
8: $\quad$ **if** $\hat{q}(S_i, A_i, \mathbf{w}) < max_a\hat{q}(S_i, \cdot, \mathbf{w})$ **then**
9: $\quad\quad$ break
10: $\quad$ **end if**
11: **end for**

The objective of replay is to propagate the reward information of the goal state to the agent's current position with fewer updates during policy evaluation. Replaying along a suboptimal trajectory takes more steps. On the one hand, to assign the credit and improve the current policy, an unbiased backup is able to help us propagate the correct value information to the agent's location with less wasteful updates than an biased backup; on the other hand, it's difficult to identify if a partial trajectory is truly optimal based off an agent's online value estimates. When the value estimates are not reliable, the resulting early break-off by unbiased replay could shorten the replayed trajectory and potentially reduce the temporal distance for passing useful value information.

The difference between unbiased and biased replay becomes more pronounced when n-step returns are used. When we perform an update, if we have replayed backward from an unbiased trajectory up to this point, we have an unbiased n-step Q-learning target under the current policy, and the update itself would likely lead to a bigger policy improvement. On the one hand, unbiased backups only explicitly propagate value information along with a select set of unbiased partial trajectories in the buffer. This could potentially lead to more bias in value function approximation and overfit a small portion of data in the buffer. On the other hand, if we have followed a trajectory mixed with suboptimal actions, the update target would be of higher variance and could potentially slow down the credit assignment process.

Chapter 4

Experiment Design

This chapter provides a detailed description of relevant design choices we made when conducting the experiments, including the selection of environments, experimental setups, and metrics used to evaluate agent performance.

4.1 Environments

Figure 4.1: Continuous Gridworld

Continuous Gridworld (ContGW) is a continuous 2-dimensional gridworld environment designed to emulate a more difficult Dyna Maze [32] under the continuous state setting. The maze environment has a width and height of length of 1 with walls on each side. At the beginning of an episode, the agent

always starts at a fixed coordinate [0, 0.5]. There are three obstacles between the starting state and the goal. If we use the left-most and right-most positions on x-axis, and the lower and upper bounds on y-axis to describe the position of the rectangle obstacle (e.g. [x_{left}, x_{right}, y_{lower}, y_{upper}]), the obstacle locations are (1) [0.2, 0.3, 0.3, 0.9], (2) [0.5, 0.6, 0.0, 0.4], and (3) [0.8, 0.9, 0.5, 1.0] respectively. The agent is given its current coordinate and allowed to take one of four actions on every step: up, down, right, and left. To simulate noisy observations an intelligent agent would naturally encounter in our physical world, each step in the environment takes the agent forward for a length of 0.05 with a $\mathcal{N}(0, 0.01)$ noise. The goal state is located in the upper right corner of the maze hiding behind an obstacle, and the agent is considered to have reached the goal when both its x and y coordinates are bigger than 0.9. We chose a sparse reward design similar to Dyna Maze so that the agent only receives a reward of +1 at the goal and 0 otherwise. The discount rate is 0.975. This domain examines how different replay methods perform under the sparse reward setting.

The challenge of this task is twofold. First, discovering the goal state is difficult, and trajectories leading up to the goal state can be noisy and rare, especially during exploration. Learning from such experience requires a replay strategy to be sample efficient. Second, learning from a sparse reward signal can be challenging since it requires credit assignment over a long temporal distance. Thus we consider ContGW a suitable domain for studying the sample efficiency property of replay algorithms.

Figure 4.2: Puddle World

Puddle world is a continuous state 2-dimensional world with $(x, y) \in [0, 1]^2$ studied by Boyan and Moore [5]. There are 2 puddles that incurs a penalty for transpassing, located at 1) [0.45, 0.4] to [0.45, 0.8] and 2) [0.1, 0.75] to [0.45, 0.75] with a radius of 0.1. The agent starts at a sampled position (x, y) where $x \in [0.1, 0.3]$ and $y \in [0.45, 0.65]$, and the navigation task ends at the goal region, which is defined as the area $x, y \in [0.95, 1.0]$. There's a cost-to-go reward of -1 per timestep. In addition, the agent also receives a penalty for transpassing the puddles calculated by $-400 \times d$ at every timestep, where d represents the distance between the agent's position and the center line of the puddles. Same as ContGW, the agent is given its current position (x, y) as observation. The agent can choose to take one action among moving up, down, left, or right at every timestep. The environment moves the agent for length 0.05 corresponding to the chosen direction, with a $\mathcal{N}(0, 0.01)$ noise. The domain applies a discount rate of 1; in other words, this environment is undiscounted. This environment highlights some challenges of credit assignment in RL algorithms since it is difficult for an agent to learn from multiple reward signals of different magnitudes (e.g., large negative rewards), which often cause the agent to erroneously decrease its value estimates too quickly and get stuck in local minima. The PuddleWorld domain has two objectives; the optimal behaviour requires the agent to learn to reach the goal from starting

region while avoiding the puddle region at the same time.

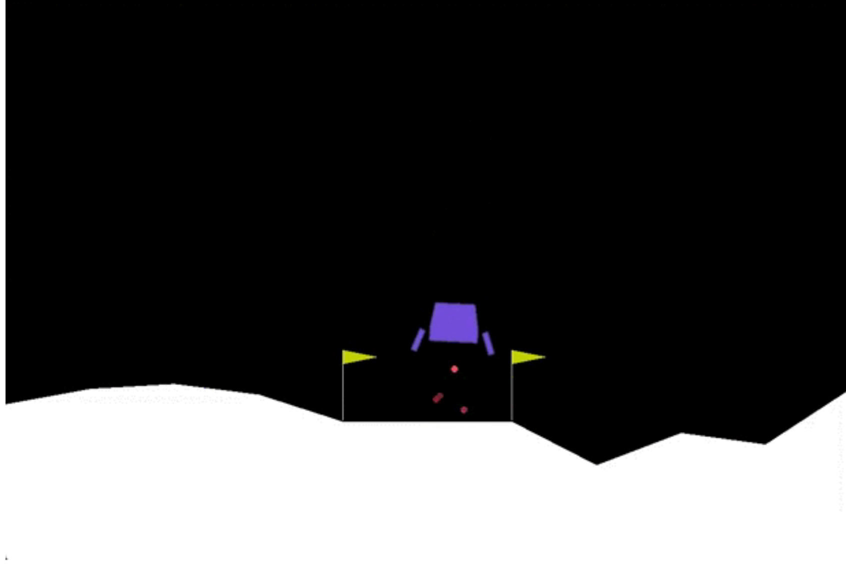

Figure 4.3: Lunar Lander

Lunar lander is a classic rocket trajectory optimization problem. We adopted the LunarLander-V2 implementation from the OpenAI gym Box2D suite. The goal of the control task is to direct the agent to the landing pad and avoid crashing. It has an 8-dimensional feature space. The observation vector describes the position, velocity and other attributes of the landing pod, such as ground contacts. The details are as shown below,

$$observation \rightarrow \begin{cases} x \quad \text{coordinate of the lander} \\ y \quad \text{coordinate of the lander} \\ v_x \quad \text{the horizontal velocity} \\ v_y \quad \text{the vertical velocity} \\ \theta \quad \text{the orientation in space} \\ v_\theta \quad \text{the angular velocity} \\ \text{Left leg touching the ground (Boolean)} \\ \text{Right leg touching the ground (Boolean)} \end{cases}$$

The coordinate values are provided relative to the landing pad, which is always at coordinates (0,0). At the beginning of an episode, the lander appears at the top center of the screen with a random initial force applied to its center of mass. Four discrete actions are available: do nothing, fire the left orientation

engine, fire the right orientation engine, and fire the main engine. And the discount rate of the environment is 0.99. A successful descent of the lander from the top of the screen to the landing pad without crashing is rewarded with around 100 to 140 points. The lander loses some reward if, instead, its landing spot is away from the landing pad. If the lander crashes, it receives an additional -100 point reward. The agent is also rewarded 10 points for a soft ground contact by each leg. Firing the main engine induces a cost-to-go reward of -0.3 points per frame, and firing the side engine takes -0.03 rewards per frame. An episode finishes when the lander crashes, moves out of the viewport or comes to rest. This domain highlights some common difficulties in replay algorithms: multiple sources of both positive and negative rewards and complex dynamics, which makes it difficult for a replay method to help the learning algorithm assign credit to the correct sequence of actions. In addition, the multimodal feature vector and high dimensional feature space make it a suitable environment to test the performance of replay methods under neural network value function approximation.

4.2 Experimental Setup

Our study explores two design dimensions in the backward replay by empirically comparing our replay agent variants in a selection of continuous state domains, where the primary concern is their early learning performance. An appropriate setup is chosen depending on the type of domains and function approximation. For example, the navigation task in ContGw and PuddleWorld experiment is relatively easy, and the experiment lasts 30k timesteps. In contrast, the feature space is high dimension and transition dynamics are more complex in LunarLander, so we let the agents run for 150k timesteps. All three domains involve some difficulty in exploration. When linear value function approximation is used in ContGW and PuddleWorld experiments, the agents are optimistically initialized to enable good exploration and follow a greedy policy. Under nonlinear function approximation, the agents execute a ϵ-greedy policy with ϵ decay. The decay rates are 0.9995, 0.9998, 0.9999

in ContGW, PuddleWorld, and LunarLander, respectively. In addition, we also employed an early episodic cutoff of 1000 steps in our puddle world and lunar lander experiments to prevent an agent from getting stuck in the environment for too long. Furthermore, the discount rate of the environments is 0.975 in ContGW, 1 or undiscounted in PuddleWorld, and 0.99 in LunarLander. In order to achieve a meaningful significance for empirical evaluation, we first sweep a combination of all hyperparameter setups using 30 independent runs. We then picked the best parameter setup for each agent variant and performed additional independent runs (between 30 and 1000) until statistical significance was achieved and learning curves were clearly separable. After each environment interaction of an agent, we first perform an online TD update to the value function, followed by the replay loop, where a number of trajectories are replayed in the reverse temporal order. A single trajectory is sampled by default when not specified. Under the tabular setting, Q-learning under function approximation could suffer from stability issues and cause soft divergence. Similar to DQN, we choose to use a target network in our experiments for better stability in training. The parameters are synced every 10 timesteps. In more recent years, bigger buffer sizes are becoming more popular, especially in deep RL applications; therefore, we choose to use a buffer size of 10k so that our results are more relevant in real-world scenarios.

4.3 Function Approximation

Tile Coding. In experiments where linear function approximation was applied, we followed Sutton's tile3.py implementation and experimented with several configurations for the fixed representation, particularly with the number of tiling in {8, 16, 32} and the number of tiles in {2, 4, 8, 16}. We chose to report the result using the overall best configuration and tile coded the coordinate observations using a setup of 16 tilings and 4 tiles, with a hash size of 1024, which is significantly larger than the max feature size of 256. The fat tiles and a large number of tilings allow the representation to possess both good generalization and discrimination suitable for the 2D nav-

igation tasks. During inference, the tile-coded representation is multiplied by a $(16 \times 4 \times 4) \times 4$ weight to compute value estimates of available actions. The weight matrix is optimistically initialized to $\frac{1}{16}$ in ContGW and 0 in PuddleWorld at the beginning of a run. During replay, the linear weight matrix is optimized using standard SGD given trajectories sampled from the experience replay buffer. The step sizes are chosen by grid search in the range of $\alpha \in \{0.03125, 0.0625, 0.125, 0.25, 0.5, 1\}$.

Neural Network. In experiments with nonlinear value function approximation, we generally found a bigger architecture is needed as the reward structure and transition dynamics become more complex. Specifically, in ContGW experiments, we discretized the raw observations (x, y coordinates) into a size 20 one-hot encoding vector for each dimension, which are then passed through a two-layer network of size [40, 16] and [16, 4]. In puddle world experiments, we followed the same feature discretization scheme but instead used a 4-layer network of size [40, 128], [128, 64], [32, 16], and [16, 4]. In the lunar lander experiment, since the features are multimodal and contain both continuous and boolean data, we relied on feature propagation of the network itself to learn a useful representation. The network follows a 3-layer architecture of [8, 128], [128, 64], and [64, 4]. The linear layers are Xavier uniform initialized [11], and ReLu nonlinearity [25] was used in between linear layers. Adam optimizer [14] was applied to update the value function parameters θ with the momentum hyperparameter $\beta \in \{0, 0.9\}$. Step sizes in the update rule are selected in the range of $\alpha \in \{0.00005, 0.00001, 0.0005, 0.0001, 0.005, 0.001\}$

4.4 Evaluation Metric

This study aims to evaluate an agent's online performance given a fixed budget of agent-environment interaction. In LunarLander, we follow the convention and evaluate the replay agents by comparing their undiscounted sum of rewards. However, in the other two maze tasks, we choose to use discounted episodic return as the evaluation metric, since it better reflects the sample efficiency of replay agents and the objective the learning algorithms optimize.

Q-learning updates the value function in order to find a policy that maximizes an agent's expected discounted return. However, there is one issue with using episodic return as a performance metric, especially with the number of timesteps being on the x-axis. An episodic return is not available until the end of an episode, but we need to report the agents' performance at every step of the episode. We resolve this issue by simply choosing the discounted return of the entire episode to be the recorded metric across all timesteps during this episode. As a result, the learning curve is a piecewise step function, where the plotted metric remains the same for every timestep within an episode. In specific, the metric G_t is defined as,

$$G_t = R_{i+1} + \gamma R_{i+2} + \cdots + \gamma^{T-1} R_{i+T},$$

with $i < t \leq i + T$, where the metric G_t is the discounted return of an episode that starts at timestep i, and T denotes the length of the episode. Note that the expected episodic return is effectively the value of the starting state.

Chapter 5

Evaluation of unbiased replay

We conducted a series of experiments to investigate the performance of unbiased replay compared to an biased replay baseline under various MDPs, representations, function approximation, and optimizer combinations. We hypothesize that unbiased replay is expected to outperform biased replay by reducing wasteful updates over suboptimal trajectories. In the first experiment, we test the agent variants in a simple continuous environment Continuous Gridworld (ContGW), where an agent is required to overcome three obstacles to reach a goal state that gives a sparse reward. The sparse reward setting is known to be a challenge for credit assignment. After that, we performed the second replay experiment in Puddle World (PuddleWorld) with cost-to-go rewards, which features a navigation task involving two objectives, reaching the goal while avoiding two puddle regions, where a good replay strategy could help balance the value propagation of both reward signals and assigns credit efficiently to complete the task with fewer samples. The large magnitude of negative reward given at the puddles also poses additional challenges to stability in training, which is a known problem in TD learning under function approximation and off-policy training. Since the domains are 2-dimensional, their state space could be nicely visualized to help our understanding of unbiased replay. In the final experiment, we evaluate the performance of unbiased replay in a Box2D environment LunarLander, which contains challenges common in real-world applications, such as high dimensional feature space and complex transition dynamics.

We evaluate the performance of unbiased replay following the order of linear and nonlinear value function approximation. In the following sections, we begin by presenting the experiment results given tile-coded representation and linear function approximation (see Section 5.1), using two continuous maze environments, ContGW and PuddleWorld. Tile coding is a coarse coding technique designed to work efficiently in multi-dimensional continuous space domains, which is applied to create fixed representations in continuous 2D state domains (one with a sparse reward and another cost-to-go). In addition, using optimistic initialization to provide good exploration, the 2D maze domains could be solved by a linear function approximator but remains a challenge in sample efficiency.

Similarly, in Section 5.2, we provide the results and analysis for an unbiased replay agent variant that uses discretized spatial features and neural network function approximation in the same maze environments as mentioned above. In addition, we take advantage of the 2D environments and visualize learned values and sampled states to better understand the different effects of the two replay strategies. Finally, we present the results of the unbiased replay agent using nonlinear function approximation in the LunarLander task. All learning curves and bar plots in our results are shown together with standard error bands for comparison.

5.1 Linear Function Approximation

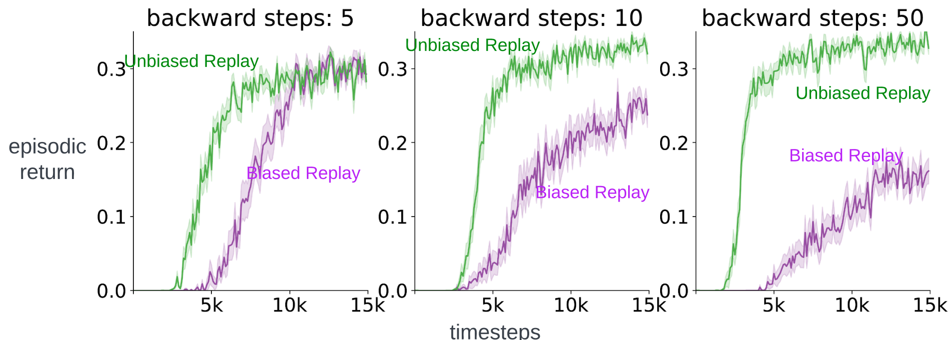

Figure 5.1: Learning curves of agents with linear function approximation in ContGW.

ContGW. To begin with, we investigate the performance of unbiased replay using a tile-coded fixed representation and linear function approximation. In Figure 5.1, we report the comparison of an unbiased replay agent and an biased replay agent in the Continuous Gridworld environment under 5, 10, and 50 backward step settings. Note the linear weights of the value function are optimistically initialized to $\frac{1}{16}$ so that the initial state action values start at 1. There is a notable difference in performance between the two replay approaches in all 3 settings. In particular, unbiased replay has shown consistently better sample efficiency when a fixed representation and linear function approximation are used to learn a value function. We also observe that as the number of backward steps increases, biased replay starts to degrade in performance in ContGW. In contrast, unbiased replay continues to benefit from more backward replay steps.

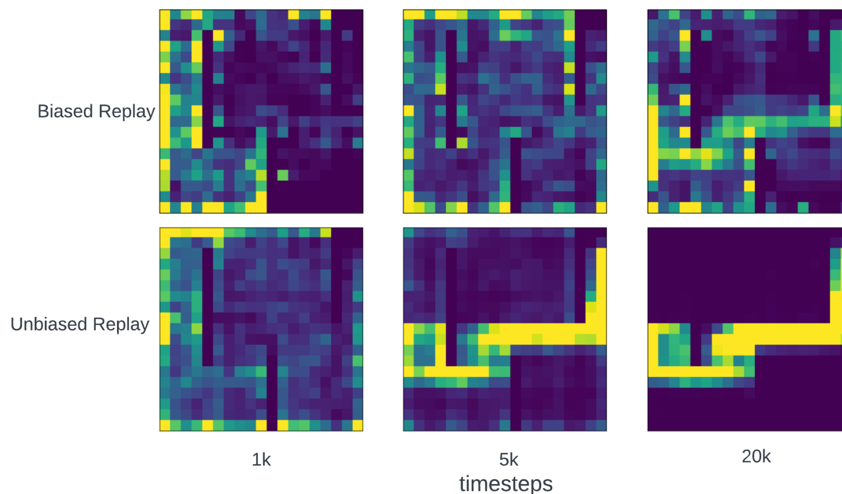

Figure 5.2: Replayed States of 50-backward-step agents with linear function approximation in ContGW (averaged per thousand timesteps).

In addition, we compared the states sampled by the two methods in Figure 5.2 under the 50-backward-step replay setting, where the brighter colour indicates the part of the state space that is being sampled more often under a replay strategy. unbiased replay starts to sample along the optimal path from the starting state to the goal state that goes around three obstacles at around 5k timestep. A similar path can be observed in the biased replay plot though until much later. It suggests that unbiased replay is able to focus limited computation over the region of interest of the state space at a much earlier stage and help the value function to converge faster. In turn, as the policy improves, the sampled trajectories are also more likely to be near-optimal; as a result, the learning process of the replay agent is more efficient.

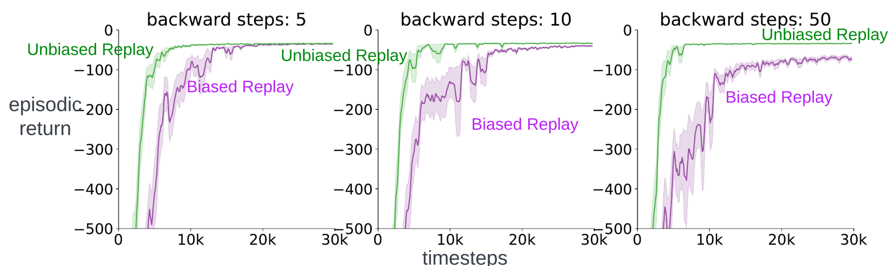

Figure 5.3: Learning curves of agents with linear function approximation in PuddleWorld. The unbiased replay agents converge to the cumulative reward of about -45.

PuddleWorld. In Figure 5.3, we perform a similar comparison between unbiased and biased replay and reach a similar conclusion. Here we have shown that unbiased replay consistently outperforms biased replay in PuddleWorld under 5, 10, and 50 replay step budget when using a tile-coded fixed representation and linear function approximation.

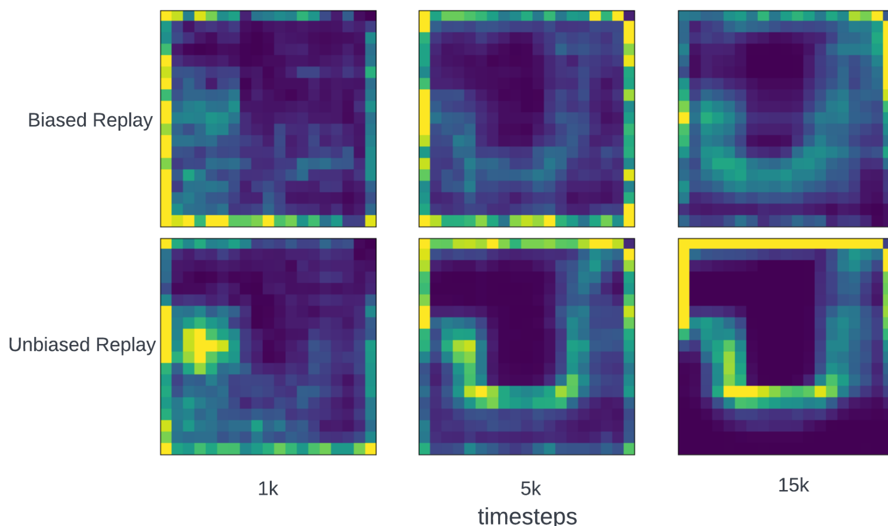

Figure 5.4: Replayed States of 50-backward-step agents with linear function approximation in PuddleWorld (averaged per thousand timesteps).

Note that since the agent starts in a small region surrounded by the puddle, there is a top and a bottom path for the agent to travel in order to reach the goal and avoid the puddle. In Figure 5.4, the comparison between their sampled states under the 50-backward-step replay setting shows a brighter colour

along two paths in the unbiased replay heatmap compared to its biased replay variant, which indicates that unbiased replay indeed samples more frequently along the optimal path towards the goal state.

Overall, in this section, we evaluate two classic replay strategies, unbiased replay and biased replay, using a fixed representation, a linear value function approximation, and optimistic initialization in two continuous state domains of sparse and cost-to-go rewards. The unbiased replay agent has shown a clear advantage in sample efficiency over the biased replay. The heatmap comparison reveals that its advantage likely comes from being able to focus a limited compute budget over the most relevant part of the state space when solving a task.

5.2 Nonlinear Function Approximation

In this section, we continue to study unbiased replay but using a learned representation and nonlinear function approximation. The agent executes an ϵ-greedy policy with an initial ϵ value of 1 and slowly decays over time to ensure enough exploration. Our linear setting utilizes a fixed representation and optimistic initialization as a comparison. Because of this, we expect the linear replay agents to converge at a faster rate with better performance in the 2-dimensional maze tasks than the nonlinear replay agents.

In addition to the two continuous 2D maze problems, we include LunaLander as a more challenging environment since its observations are multimodal and no longer discretized before being taken into the value network as input. In addition, the environment comes with more complex transition dynamics and its state space is high dimensional; therefore, a good representation and state abstraction are hard to attain. We believe this small Box2D simulation domain is more representative of real-world control problems than the maze domains and would like to see if unbiased replay would still perform well under this environment.

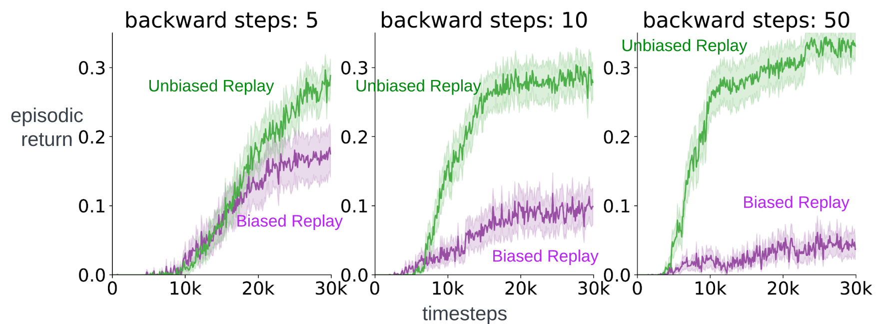

Figure 5.5: Learning curves of agents with nonlinear function approximation in ContGW.

ContGW. The learning curve plot in Figure 5.5 shows that unbiased replay converges faster than biased replay in ContGW under 5, 10, and 50 replay steps when a neural network is used to approximate the value function. As the number of backward steps extends, biased replay suffers from visible degradation in performance.

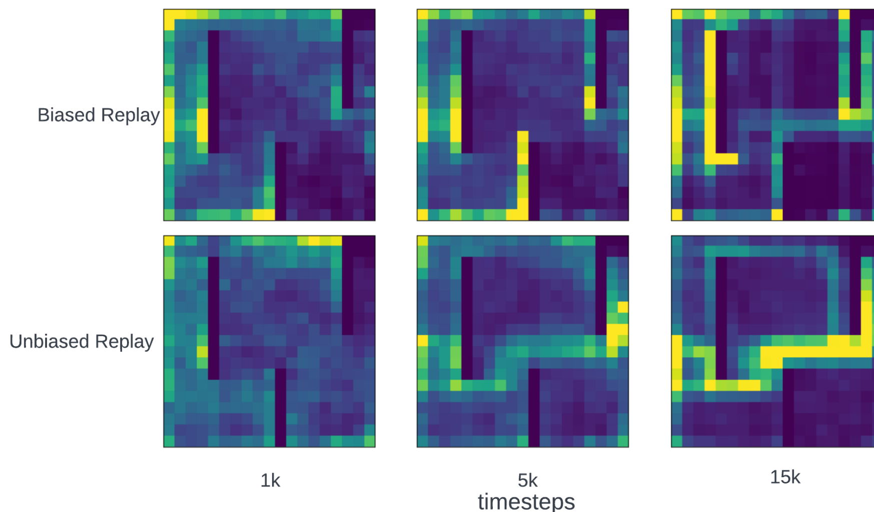

Figure 5.6: Replayed States of 50-backward-step agents with nonlinear function approximation in ContGW (averaged per thousand timesteps).

In the replayed state heatmap under the 50-backward-step replay setting, the sampling distribution under biased replay is more scattered. In contrast, unbiased replay is able to sample along the optimal paths quite notably after

5k timestep.

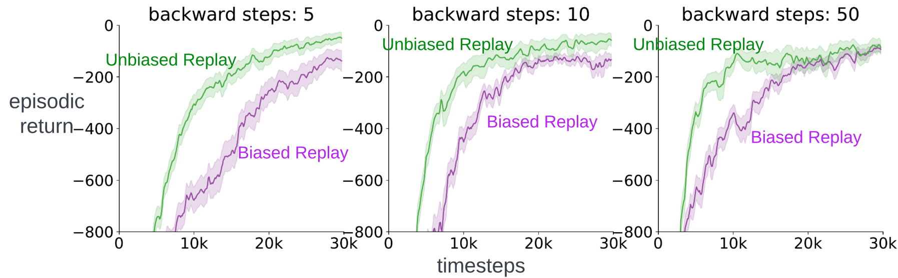

Figure 5.7: Learning curves of agents with nonlinear function approximation and an exploratory policy in PuddleWorld.

PuddleWorld. Figure 5.7 shows the comparison of unbiased v.s. biased replay given a good exploratory behaviour policy. Because of this, and also the fact that the neural network function approximation requires learning a representation, agents learn slower compared to the linear variants, which uses optimistic initialization and a fixed representation. The agent in this experiment first goes through an exploration phase with an initial ϵ value of 1, which slowly decays over time. In this case, unbiased replay demonstrates a clear advantage over biased replay's sample efficiency when comparing their learning curves, similar to the linear function approximation result in PuddleWorld.

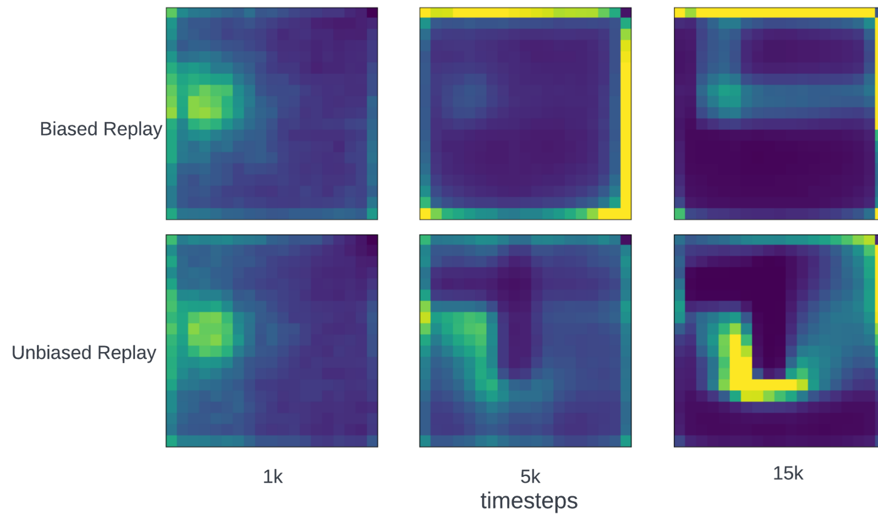

Figure 5.8: Replayed states of 50-backward-step agents with nonlinear function approximation and an exploratory policy in PuddleWorld (averaged per thousand timesteps)

This is no surprise when we look into the states sampled for replay in Figure 5.8. Even though in the beginning, both methods sample a lot from the starting region of the maze, as time passes, the biased replay heatmap indicates a strong focus is being placed over rather suboptimal trajectories that transpasses the puddle instead of going around it, while unbiased replay is able to sample heavily over the more pertinent part of the state space, where the agent avoids the puddle to reach the goal state.

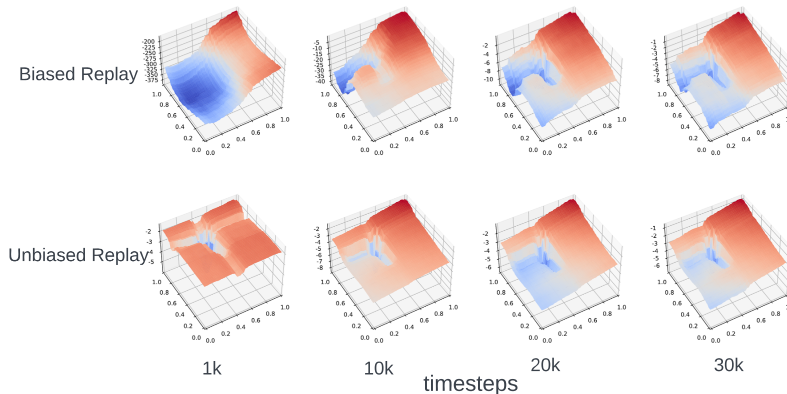

Figure 5.9: Learned values of 50-backward-step agents with nonlinear function approximation and an exploratory policy in PuddleWorld

The value maps largely corroborate the same story. The value function learned by biased replay in Figure 5.9 shows a slow progression and poor generalization. In contrast, the value function learned by unbiased replay converges rather quickly, where the blue puddle region is visible almost from the start. And despite overgeneralizing to neighbouring states in the beginning, it quickly learns to correctly identify the puddle region that helps direct the agent to reach the goal while avoiding the puddle.

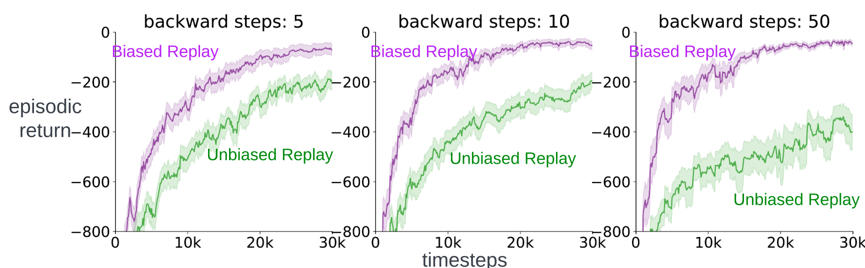

Figure 5.10: Learning curves of agents with nonlinear function approximation and a greedy policy in PuddleWorld.

Next, we continue to test the limits of unbiased replay by investigating a pathological replay setting where we replace the exploratory behaviour policy with a greedy policy. Contrary to previous results, biased replay exhibits a consistent better sample efficiency than unbiased replay across 5, 10, and 50

backward steps when a greedy behaviour policy is applied, as shown by the learning curves in Figure 5.10. Furthermore, the online performance of the unbiased replay agent degrades as the number of backward steps increases. The 50 backward step setting also shows a clear sign of the unbiased replay agent converging to a suboptimal policy.

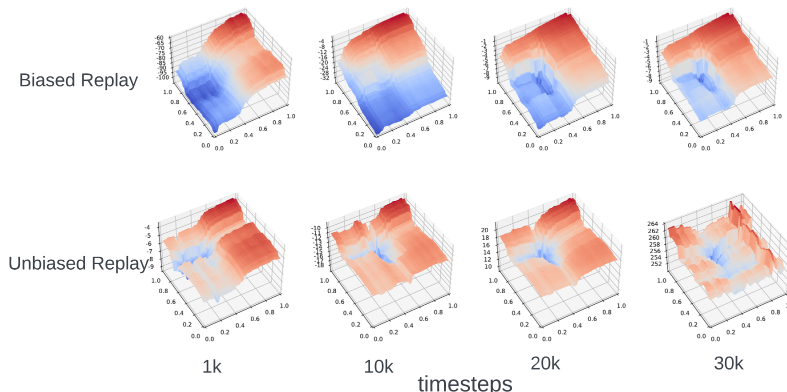

Figure 5.11: Learned values of 50-backward-step agents with nonlinear function approximation and a greedy policy in PuddleWorld.

This observation is confirmed by looking at the value function being learned under unbiased replay, where signs of divergence are shown as degradation, and poor generalization is visible among the state values in Figure 5.11. In the meantime, despite the value function learned by biased replay seemingly being unable to converge quickly, it has a smooth surface, where a clear light blue puddle region is recognizable from around 20k timestep. It shows that the puddle is slowly but correctly identified, and a good state generalization is achieved under biased replay.

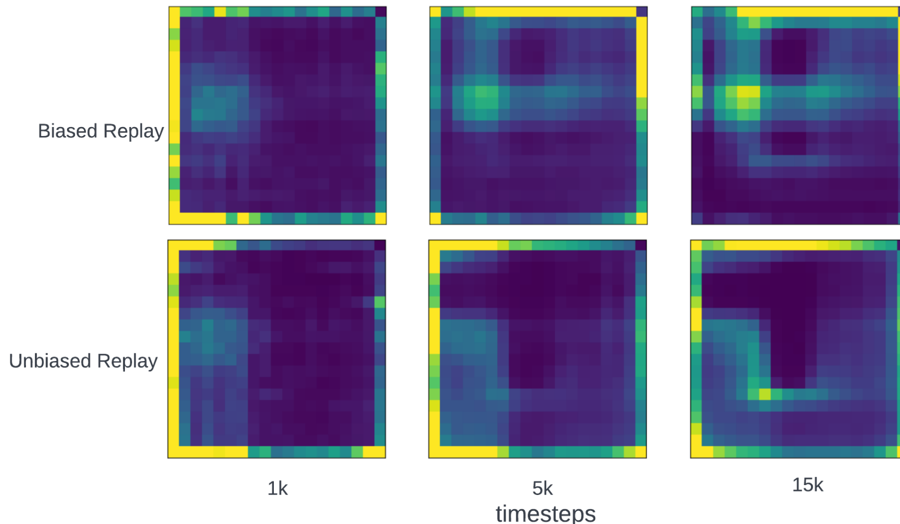

Figure 5.12: Replayed states of 50-backward-step agents with nonlinear function approximation and a greedy policy in PuddleWorld

When we look further into the sampled states under the two replay strategies, it is not difficult to understand why. For example, figure 5.12 reveals that biased replay samples are often along the optimal trajectory, as indicated by the bright yellow colour starting from 5k timestep. And even though the biased replay sample along suboptimal trajectories that crosses the puddle sometimes, which could explain its slow convergence, it avoids the highest penalty region where the two puddle overlaps. In comparison, even though the states that unbiased replay samples successfully avoid the puddle, unbiased replay also has seemingly sampled along many suboptimal trajectories in the lower part of the maze, which could lead to wasteful updates that could explain the underperformance.

Interestingly, the value function learned under a greedy policy converges faster than the one under the exploratory policy. It could be partially explained by the fact that the behaviour policy influences the state distribution of the buffer that, in turn, impacts the replay state distribution. This could be seen by the replayed states since it is uniformly sampled from the buffer under biased replay. The sampling distribution of biased replay under a greedy policy in Figure 5.12, compared to its exploratory policy counterpart from Figure 5.8, is able to sample more often along trajectories that avoid the puddle. This

has a similar effect on the sampling distribution with unbiased replay.

We hypothesize the following reasons for the underperformance of unbiased replay under a greedy behaviour policy. First, it is likely that the greedy policy of the agent changes more quickly compared to an exploratory policy in the beginning. Therefore, state transitions along the optimal path likely have low coverage in the buffer during this time as state distribution of the buffer may be sporadic and highly skewed. As a result, the additional bias from the unbiased replay updates could harm the stability of TD learning and even lead to soft divergence, as observed in the learned value heatmap. Second, large negative rewards are given at the puddles of the environment, it is easy for a greedy agent to learn to avoid the huge penalty by running away from the puddle, but the challenge of this environment is to avoid getting stuck in this local minima and learn to navigate to the goal state. It is likely that, at first, the trajectories leading into the goal state are highly suboptimal. In such a case, unbiased replay could have been distracted from propagating the value information of the goal state to the starting state region but focusing too much on learning to avoid the puddle since unbiased replay could choose to break off early along a suboptimal trajectory.

In this experiment, we expected unbiased replay to outperform biased replay, the same as before, which was not the case. Our work only takes the first step to showcasing this empirical result; future work can be done to help understand and explain the phenomenon. We observed that the performance of unbiased agents is still improving, but the value function learned sometimes shows signs of divergence. Since we mostly care about early performance, we observed that contrary to our initial expectation, unbiased replay underperforms biased replay in this setting. We are confident of the reliability of the results since, as a comparison, the biased agent variants are converging and replaying shorter trajectories also shows better performance in Figure 5.10.

LunaLander. To better understand unbiased replay's performance beyond 2D maze problems, we experimented with a more difficult control task LunarLander. The goal of the task is to direct the agent to reach the landing pad as gently and fuel-efficiently as possible. The challenge is the observa-

tion space, which contains 5 continuous state variables and 2 boolean-type variables. As a result, the observation space is immense, considering the transition dynamics are fairly complex. A neural network is designed to consume multimodal feature vectors and solve the curse of dimensionality problem, which makes it a great fit for testing the performance of unbiased replay in the LunarLander task, compared to using tile-coding and linear function approximation.

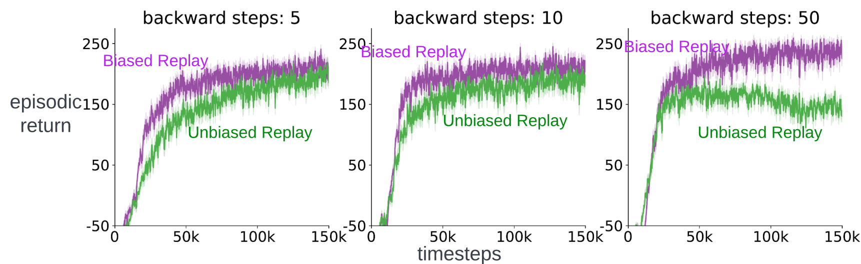

Figure 5.13: Learning curves of agents with nonlinear function approximation in LunarLander.

Figure 5.13 shows the learning progression of the two replay strategies in LunarLander. biased replay outperforms unbiased replay in all replay settings, even though the agent begins with a fully exploratory policy. We hypothesize several possible reasons behind the phenomenon. First, unbiased replay relies on relatively accurate value estimates. In LunarLander, since the input features are multimodal and transition dynamics are higher dimensional compared to the maze tasks, it could take some time for the neural network to learn a good representation and generate good value estimates. This could lead to replaying many suboptimal trajectories before focusing on good ones and slows down the learning process. Second, the main reward or penalty for a landing in the LunarLander task is given at the very end of an episode even though a mistake could be made by an agent very early on and leads to a crash; therefore, it may be more beneficial, especially in the beginning to simply replay a trajectory as further back as allowed by the compute budget to propagate credit to an earlier mistake or a critical move for a successful

landing. Replaying trajectories only following on-policy actions may shorten the temporal distance when assigning credit and leads to a slower convergence. Finally, since early exploration could easily crash the lander, the buffer is filled with highly suboptimal trajectories. It is reasonable to assume that initially, the buffer has a low and skewed coverage in the state space. By preferentially replaying on-policy transitions under such a skewed sample distribution, the learning process may suffer from high sampling bias that incorrectly overestimates some states and harms the agent's performance, and not able to harness the benefit from prioritizing on-policy state transitions in the useful region of the state-action space to achieve a good landing.

5.3 Analysis of Agent Performance and Replay Step Budget

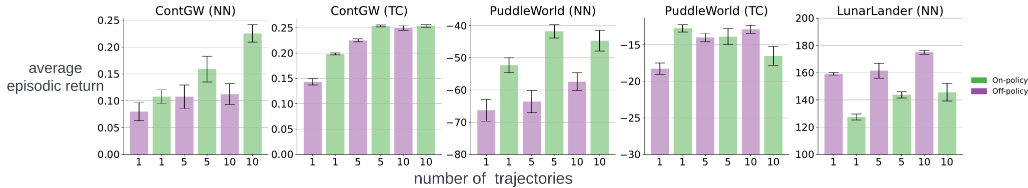

Figure 5.14: Average return under a different number of replayed trajectories per timestep.

In the last subsection, we examine the scaling performance of unbiased and biased replay. Specifically, we simulate a number of computation budget settings by varying the number of replay steps from 5 to 50, by either sampling more trajectories or longer trajectories. On the one hand, when short 5-step trajectories are being replayed, as is shown in Figure 5.14, increasing the number of sampled trajectories mostly improves an agent's sample efficiency in 9 out of 10 replay settings, including both replay strategies. The only exception happens with the unbiased replay agent variant under linear function approximation in PuddleWorld, where its performance peaks between replaying 1 and 5 trajectories and worsens when the number of sampled trajectories increases

to 10.

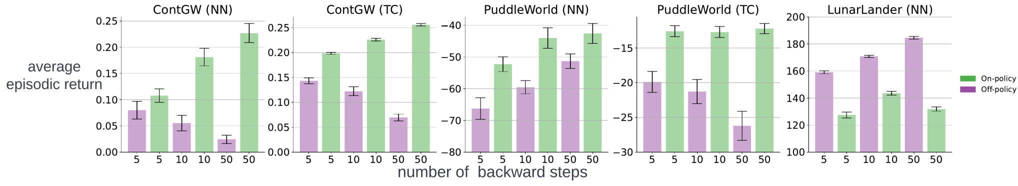

Figure 5.15: Average return under different number of backward steps.

On the other hand, an observation in Figure 5.15 is that unbiased replay tends to scale better when the additional compute is used to extend the number of backward steps. In specific, biased replay's online performance only improves in 2 out of 5 settings, while unbiased replay generally shows a similar or better performance when longer trajectories are being replayed except for a regression in the 50-step setting in LunarLander.

5.4 Conclusions

Based on the above experiment results, we observe that,

1. given good exploration, unbiased replay outperforms biased replay in two continuous state 2D maze domains. This includes sparse reward and cost-to-go reward MDPs, where both linear and nonlinear function approximations were tested,

2. biased replay agent using nonlinear function approximation shows a better online performance in PuddleWorld when a greedy policy was applied,

3. biased replay agent using nonlinear function approximation outperforms the unbiased replay agent variant in the LunarLander domain,

4. when the additional computation is allowed, increasing the number of sampled trajectories generally helps improve agent performance, while unbiased replay tends to benefit more consistently given more backward steps.

Chapter 6

Jumpy Replay

TD learning is a slow process, especially when value information has to be propagated through a long action sequence to learn a good policy. An efficient search can be done by working backward timestep by timestep and updating state values along the way. Given a fixed computation budget, a sample efficient backward replay strategy would be to replay across a temporal distance as far back as possible.

A related background is what constitutes a good value function approximation in one-step TD learning. A natural intuition would suggest that $\tilde{V}(s)$ should be close to the optimal value function $V^*(s)$ everywhere; however, this is not necessary to achieve good suboptimal control. For instance, if $\tilde{V}$ differs from V^* uniformly by a constant over all states, the resulting policy from $\tilde{V}$ would still be optimal. This suggests an alternative condition for a good value function approximation in suboptimal control, where a good policy could be achieved as long as the difference of $\tilde{V}$ and V^* are close for all pairs of states s and s', instead of the state values themselves,

$$\tilde{V}(s) - \tilde{V}(s') \approx V^*(s) - V^*(s').$$

Similarly, for state-action values, as long as the approximation error $Q(s,a) - \hat{Q}(s,a)$ of a state s changes gradually with respect to actions, the resulting policy would achieve similar control performance [4].

Furthermore, as a result of function approximation and state generalization, a value update could affect a small region in the state space that covers more than a single point estimate. Thus a good policy could be learned before

all state value converges to optimal values. Instead, we could replay a trajectory while skipping some updates in between to propagate value information over a longer temporal distance given the same number of updates, as long as the learned value function maintains a similar distance from $V^*(s)$ across all states. In essence, changing the order of updates is not expected to negatively affect the quality of function approximation, provided all transitions are visited a similar number of times.

Here we introduce the idea of jumpy replay, a simple time-stepping replay strategy where some replay steps are skipped so that the same computation budget can be used to replay a longer trajectory. This would not have been ideal when used with tabular methods (e.g., tabular Dyna-Q). Without state generalization, the backward focusing principle tells us that a useful update of a state value relies on value information propagated from its direct successors [32]. Therefore a trajectory update would be better off by strictly following a reverse temporal order than skipping some steps in between. However, with good function approximation, such as a neural network, a generalizing state representation could be learned even in a continuous state setting. As a result, an action value update could improve value estimates in a small region of its neighbourhood in the state space. If a trajectory contains temporally consecutive state transitions that are highly generalizable, it could potentially enable faster temporal credit assignment through replaying in a more sporadic fashion over such a sampled trajectory. Figure 6.1 shows an example MDP where jumpy replay could potentially speed up value propagation over a long trajectory, where states selected for the replay are labelled in green.

Furthermore, normally one-step TD methods use the same time step for how often the action can be updated and the timescale over which bootstrapping is done. Like n-step methods help select the time interval of bootstrapping, jumpy replay provides more flexibility for the timescale of action updates. In applications requiring less frequent change in actions, jumpy replay is expected to speed up temporal credit assignment and improve sample efficiency. In addition, replaying trajectories at a lower temporal granularity could also potentially help reduce correlation and interference between consecutive up-

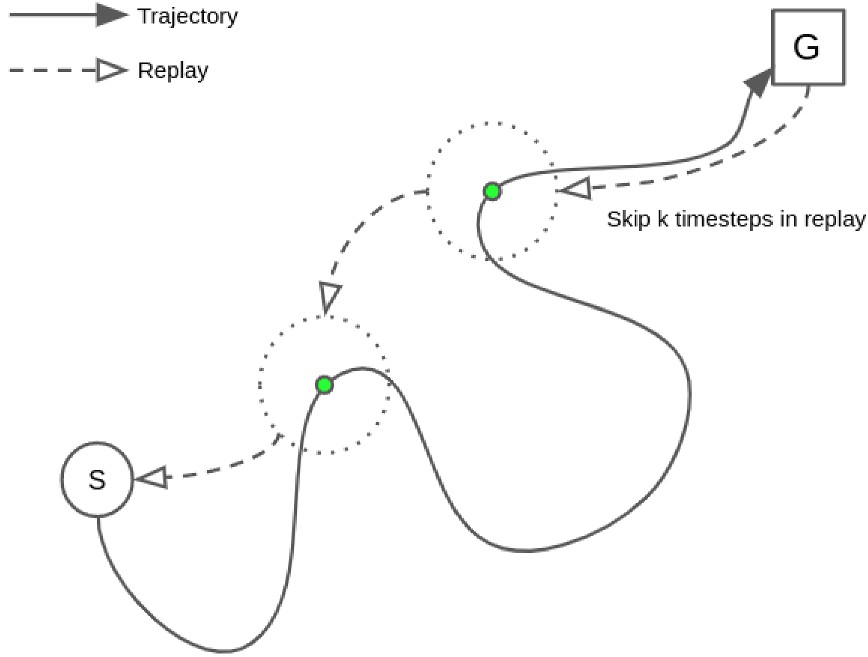

Figure 6.1: An example of time-stepping mechanic in jumpy replay. The solid line depicts a suboptimal trajectory with detours. The dashed line shows the concept of jumpy replay where a predefined number of k timesteps are skipped between each replay step.

dates. Algorithm 4 shows the pseudocode of jumpy replay.

Algorithm 4 Jumpy Replay (one-step return)

1: Initialize number of updates m
2: Initialize number of steps to skip between updates k
3: Loop repeat n times:
4: Sample a trajectory : $S_0, A_0, R_1, \ldots, S_{(m-1)\times k+1}, A_{(m-1)\times k+1}$
5: **for** $i = (m-1)\times k, (m-2)\times k, \ldots, k, 0$ **do**
6: $\quad \delta_i = R_{i+1} + \gamma max_a \tilde{q}(S_{i+1}, \cdot, \mathbf{w}) - \hat{q}(S_i, A_i, \mathbf{w})$
7: $\quad$ Update $\mathbf{w}$ with $\nabla(\delta_i^2)$ for a step size η
8: **end for**

6.1 Empirical Results

Here we investigate the empirical performance of jumpy replay given 10 replay step budgets per time step and present some first results under a diverse set of domains and function approximations by comparing jumpy replay agents

using a different number of skip steps with a one-step backward replay baseline. Note unlike the classic replay experiments in the previous chapter, the backward replay baseline does not utilize n-step returns and generally requires more replay steps to learn a task, hence we ran the experiments longer to account for this. We hypothesize that given good function approximation, jumpy replay with an appropriate number of skip steps is expected to outperform the backward replay (1-step jump) baseline in various domains. The learning curves are plotted together with standard error bands for comparison.

6.1.1 Linear Function Approximation

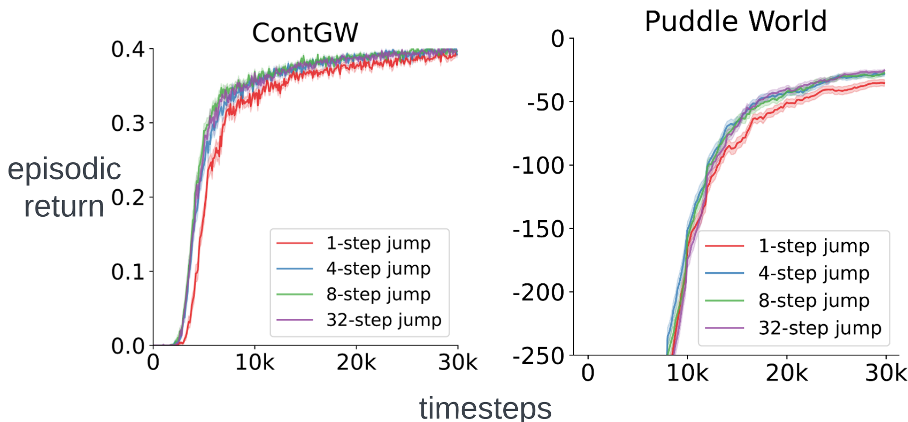

Figure 6.2: Learning curves of 10-backward-step jumpy replay agents with linear function approximation.

Figure 6.2 showcases the comparison between jumpy replay variants with a different number of skipped timesteps. Note the one-step backward replay baseline is equivalent to vanilla backward replay, in which learning curves are labelled in red. Under the use of tile-coded fixed representation and linear value function approximation, jumpy replay variants outperform their one-step jump baselines in ContGW and PuddleWorld by a small margin. The results indicate that it is possible to take advantage of state generalization to speed up credit assignment in continuous state domains when a linear value function approximation is used.

6.1.2 Nonlinear Function Approximation

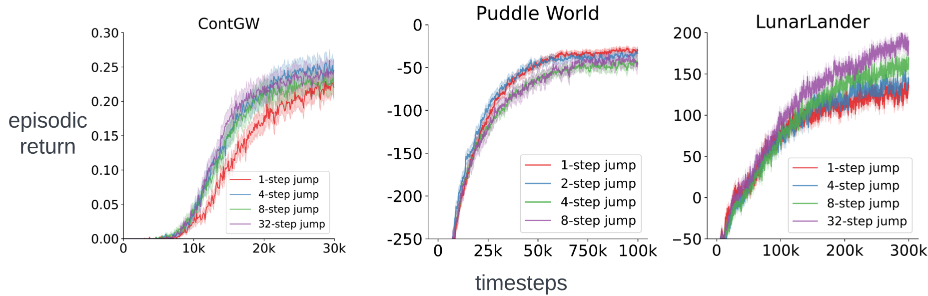

Figure 6.3: Learning curves of 10-backward-step jumpy replay agents with nonlinear function approximation.

Figure 6.3 shows the learning curves of agent variants under test in 3 continuous state domains under nonlinear function approximation, where jumpy replay variants have shown an improvement in performance in 2 out of 3 domains. It is worth noting that jumpy replay improves the replay agent's sample efficiency quite notably in LunarLander, where action could have a long-lasting effect on the future outcome. For example, a bad action that topples the lander at the beginning of an episode could lead to a catastrophic crash after hundreds of timesteps and receives a large negative reward at the end of an episode. Similar improvement is observed in ContGW, where key actions and the reward given at the goal state could be distant. Our result suggests that jumpy replay could be more advantageous than sequential backward replay in such scenarios. However, the jumpy replay strategy does not appear to provide a benefit in PuddleWorld. More specifically, with a larger jump size of 32 timesteps, the jumpy replay agent variant underperforms the baseline in PuddleWorld, while jumpy replay agent variants using a jump size of 4 and 8 timesteps do not significantly outperform the baseline either.

We hypothesize the following possible reasons. First, most performance improvement in PuddleWorld is from avoiding the puddle next to the agent's starting region. The large penalty for transpassing the puddle region does not require credit assignment over a long temporal distance for an agent to learn a policy to avoid the puddle. Thus jumpy replay might be of little benefit in

such an MDP setting. Second, it may have taken some time to learn a good state representation using a neural network before we could take advantage of good state generalization. As a result, performance improvement through the time-stepping mechanic could be less than using a fixed representation.

Despite being able to skip updates by taking advantage of good function approximation, jumpy replay has several limitations. First, it introduces an additional hyperparameter to search the number of skipping timesteps k. Since the parameter choice of k is likely problem-dependent, we would need to perform a parameter search for every new problem encountered. Second, the advantage of jumpy replay relies heavily on the quality of state generalization and value function approximation, which can be difficult to assess using a quantitative test. On top of it, a good parameter value k would also likely change over time as a value function receives more updates and the optimality of sampled trajectories improves.

In conclusion, jumpy replay variants exhibit a comparable or better sample efficiency compared to the backward replay baseline in our experiments. It has not been shown to hurt an agent's online performance in all of our experiments, and it sometimes helps improve significantly over the backward replay baselines.

Chapter 7

Conclusion and Future Work

Experience replay is an effective way of improving sample efficiency in model-free RL. Developing better replay strategies that could take advantage of large-scale function approximation and overcome the challenges from correlation in data and off-policy updates could be a promising way to further improve the learning efficiency of online agents.

In this thesis, we take the first few steps towards understanding the utility of select replay strategies building on top of the understudied classic backward replay. First, we contribute by making an effort to better understand the effect of the unbiased replay strategy proposed in the original experience replay literature. We empirically evaluate the performance of unbiased v.s. biased backward replay in several classic RL environments using different function approximators and representations and observe that unbiased replay outperforms biased replay in ContGW and PuddleWorld significantly when a good exploratory policy is present. However, we can also find two scenarios where unbiased replay leads to an inferior online performance under the use of nonlinear function approximation. More specifically, when a greedy policy is applied from the get-go in PuddleWorld, using an unbiased replay strategy appears detrimental compared to the biased replay agent baseline. Furthermore, the unbiased replay agent using nonlinear function approximation underperforms its biased replay baseline in the LunarLander environment. Overall, the benefit of unbiased replay appears to be problem dependent and sensitive to the state distribution of the buffer.

In the meantime, there are some limitations to our work on unbiased replay. One limitation of unbiased replay is that its performance sometimes suffers from sampling bias. Possible remedies that could be tested in the future include importance sampling correction or combining unbiased replay with TD algorithm variants that are more robust to off-policy learning, such as QRC [10]. Another limitation is that all of our experiments are performed under the discrete action setting and use q-learning for TD control. As for the next steps, similar replay experiments could also be carried out under continuous action control and extended to policy gradient methods in order to better understand the generality of our observations. Finally, this work mainly focuses on conducting experiments in smaller domains when studying the replay strategies in question. This is due to the availability of computing resources and our goal to achieve meaningful experiment results with statistical significance through investigating multiple replay settings and contracting a large number of independent runs. It remains to be seen what advantages these replay strategies could bring to the table in bigger domains and more challenging tasks.

Second, we propose a time-stepping replay strategy called jumpy replay, which is designed to take advantage of state generalization under function approximation and propagate value information further along a trajectory given a limited compute. We empirically compare jumpy agent variants using various jump sizes and observe that, with the appropriate choice of jump size, jumpy replay has shown a comparable or better performance consistently across 5 replay settings under both linear and nonlinear function approximation compared to a vanilla backward replay baseline. We also found that the agent enjoys a bigger improvement in ContGW and LunarLander compared to the PuddleWorld environment. Thus we conclude that replaying trajectories in the reverse temporal order while skipping some state transitions can save limited computation and speed up credit assignment in some domains.

There are several limitations to the jumpy replay strategy. First, it introduces an additional hyperparameter, the number of state transitions to skip k, which the best parameter value is also likely to change throughout an agent's

lifetime. To alleviate the problem, the appropriate number of skip steps could be learned as a function of state instead of a constant value chosen from a hyperparameter search. Second, jumpy replay by design relies heavily on the quality of state generalization, which is not always guaranteed. Instead, other regularized representations such as the Laplacian [41] could be applied to help jumpy replay take advantage of better generalization across states.

Overall, the empirical results of jumpy replay could inspire future directions in developing more sample-efficient techniques in subsampling or augmenting trajectories in experience replay and possibly in model-based planning, where state generalization could be leveraged for efficient temporal credit assignment. For example, real experience could be used to guide policy improvement and replay could be integrated as a part of the planning process. In addition, it is possible that similar replay techniques could also be applied to speed up learning in general value functions [40].

References

[1] R Ellen Ambrose, Brad E Pfeiffer, and David J Foster. "Reverse replay of hippocampal place cells is uniquely modulated by changing reward." In: *Neuron* (2016).

[2] Andrew Gehret Barto, Steven J Bradtke, and Satinder P Singh. *Real-time learning and control using asynchronous dynamic programming.* University of Massachusetts at Amherst, Department of Computer and Information Science, 1991.

[3] Richard Bellman. "Dynamic programming." In: *Science* (1966).

[4] Dimitri Bertsekas. *Reinforcement learning and optimal control.* Athena Scientific, 2019.

[5] Justin Boyan and Andrew Moore. "Generalization in reinforcement learning: Safely approximating the value function." In: *Advances in neural information processing systems* (1994).

[6] Romain Cazé, Mehdi Khamassi, Lise Aubin, and Benoıt Girard. "Hippocampal replays under the scrutiny of reinforcement learning models." In: *Journal of neurophysiology* (2018).

[7] Peng Dai and Eric A Hansen. "Prioritizing Bellman Backups without a Priority Queue." In: *International Conference on Automated Planning and Scheduling.* 2007.

[8] William Fedus, Prajit Ramachandran, Rishabh Agarwal, Yoshua Bengio, Hugo Larochelle, Mark Rowland, and Will Dabney. "Revisiting fundamentals of experience replay." In: *International conference on machine learning.* Proceedings of Machine Learning Research. 2020.

[9] Stuart Geman, Elie Bienenstock, and René Doursat. "Neural networks and the bias/variance dilemma." In: *Neural computation* (1992).

[10] Sina Ghiassian, Andrew Patterson, Shivam Garg, Dhawal Gupta, Adam White, and Martha White. "Gradient temporal-difference learning with regularized corrections." In: *International conference on machine learning.* Proceedings of Machine Learning Research. 2020.

[11] Xavier Glorot and Yoshua Bengio. "Understanding the difficulty of training deep feedforward neural networks." In: *The thirteenth international conference on artificial intelligence and statistics*. Journal of Machine Learning Research. 2010.

[12] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep learning*. MIT press, 2016.

[13] Taher Jafferjee, Ehsan Imani, Erin Talvitie, Martha White, and Micheal Bowling. "Hallucinating value: A pitfall of dyna-style planning with imperfect environment models." In: *arXiv preprint arXiv:2006.04363* (2020).

[14] Diederik P Kingma and Jimmy Ba. "Adam: A method for stochastic optimization." In: *International Conference on Learning Representations*. 2015.

[15] George Konidaris, Sarah Osentoski, and Philip Thomas. "Value function approximation in reinforcement learning using the Fourier basis." In: *Twenty-fifth AAAI conference on artificial intelligence*. 2011.

[16] Derek Li, Andrew Jacobsen, and Adam White. "Revisiting experience replay in non-stationary environments." In: *International conference on autonomous agents and multiagent systems* (2021).

[17] Yitao Liang, Marlos C Machado, Erik Talvitie, and Michael Bowling. "State of the art control of atari games using shallow reinforcement learning." In: *arXiv preprint arXiv:1512.01563* (2015).

[18] Timothy P Lillicrap, Jonathan J Hunt, Alexander Pritzel, Nicolas Heess, Tom Erez, Yuval Tassa, David Silver, and Daan Wierstra. "Continuous control with deep reinforcement learning." In: *arXiv preprint arXiv:1509.02971* (2015).

[19] Long Ji Lin. "Programming robots using reinforcement learning and teaching." In: *AAAI conference on artificial intelligence*. 1991.

[20] Long-Ji Lin. "Self-improving reactive agents based on reinforcement learning, planning and teaching." In: *Machine learning* (1992).

[21] James L McClelland, David E Rumelhart, and Geoffrey E Hinton. "The appeal of parallel distributed processing." In: *MIT Press, Cambridge MA* (1986).

[22] Volodymyr Mnih, Adria Puigdomenech Badia, Mehdi Mirza, Alex Graves, Timothy Lillicrap, Tim Harley, David Silver, and Koray Kavukcuoglu. "Asynchronous methods for deep reinforcement learning." In: *International conference on machine learning*. Proceedings of Machine Learning Research. 2016.

[23] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. "Playing atari with deep reinforcement learning." In: *arXiv preprint arXiv:1312.5602* (2013).

[24] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A Rusu, Joel Veness, Marc G Bellemare, Alex Graves, Martin Riedmiller, Andreas K Fidjeland, Georg Ostrovski, Stig Petersen, Charles Beattie, Amir Sadik, Ioannis Antonoglou, Helen King, Dharshan Kumaran, Daan Wierstra, Shane Legg, and Demis Hassabis. "Human-level control through deep reinforcement learning." In: *Nature* (2015).

[25] Vinod Nair and Geoffrey E. Hinton. "Rectified linear units improve restricted boltzmann machines." In: *International conference on machine learning*. 2010.

[26] Yangchen Pan, Muhammad Zaheer, Adam White, Andrew Patterson, and Martha White. "Organizing experience: a deeper look at replay mechanisms for sample-based planning in continuous state domains." In: *arXiv preprint arXiv:1806.04624* (2018).

[27] Ronald Parr, Lihong Li, Gavin Taylor, Christopher Painter-Wakefield, and Michael L Littman. "An analysis of linear models, linear value-function approximation, and feature selection for reinforcement learning." In: *International conference on machine learning*. 2008.

[28] Tom Schaul, John Quan, Ioannis Antonoglou, and David Silver. "Prioritized experience replay." In: *arXiv preprint arXiv:1511.05952* (2015).

[29] David Silver, Aja Huang, Chris J Maddison, Arthur Guez, Laurent Sifre, George Van Den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, Sander Dieleman, Dominik Grewe, John Nham, Nal Kalchbrenner, Ilya Sutskever, Timothy Lillicrap, Madeleine Leach, Koray Kavukcuoglu, Thore Graepel, and Demis Hassabis. "Mastering the game of Go with deep neural networks and tree search." In: *Nature* (2016).

[30] Peter Stone, Richard S Sutton, and Gregory Kuhlmann. "Reinforcement learning for robocup soccer keepaway." In: *Adaptive Behavior* (2005).

[31] Richard S Sutton. "Learning to predict by the methods of temporal differences." In: *Machine learning* (1988).

[32] Richard S Sutton and Andrew G Barto. *Reinforcement learning: An introduction*. MIT press, 2018.

[33] Richard S Sutton, Csaba Szepesvári, Alborz Geramifard, and Michael P Bowling. "Dyna-style planning with linear function approximation and prioritized sweeping." In: *arXiv preprint arXiv:1206.3285* (2012).

[34] Richard Stuart Sutton. "Temporal credit assignment in reinforcement learning." PhD thesis. University of Massachusetts Amherst, 1984.

[35] Hado Van Hasselt, Yotam Doron, Florian Strub, Matteo Hessel, Nicolas Sonnerat, and Joseph Modayil. “Deep reinforcement learning and the deadly triad.” In: *arXiv preprint arXiv:1812.02648* (2018).

[36] Hado P Van Hasselt, Matteo Hessel, and John Aslanides. “When to use parametric models in reinforcement learning?” In: *Advances in neural information processing systems* (2019).

[37] Harm Van Seijen and Rich Sutton. “Planning by prioritized sweeping with small backups.” In: *International conference on machine learning*. Proceedings of Machine Learning Research. 2013.

[38] Ziyu Wang, Victor Bapst, Nicolas Heess, Volodymyr Mnih, Remi Munos, Koray Kavukcuoglu, and Nando de Freitas. “Sample efficient actor-critic with experience replay.” In: *arXiv preprint arXiv:1611.01224* (2016).

[39] Christopher JCH Watkins and Peter Dayan. “Q-learning.” In: *Machine learning* (1992).

[40] Adam White. “Developing a predictive approach to knowledge.” PhD thesis. University of Alberta, 2015.

[41] Yifan Wu, George Tucker, and Ofir Nachum. “The laplacian in rl: Learning representations with efficient approximations.” In: *International Conference on Learning Representations*. 2019.

[42] Yi Xu, Qi Qian, Hao Li, and Rong Jin. “Why Does Multi-Epoch Training Help?” In: *arXiv preprint arXiv:2105.06015* (2021).

[43] Shangtong Zhang and Richard S Sutton. “A deeper look at experience replay.” In: *arXiv preprint arXiv:1712.01275* (2017).